Manta: Hybrid-Sensitive Type Inference Toward Type-Assisted Bug Detection for Stripped Binaries

Chengfeng Ye
cyeaa@cse.ust.hk
The Hong Kong University of Science and Technology
China

Yuandao Cai*
ycaibb@cse.ust.hk
The Hong Kong University of Science and Technology
China

Anshunkang Zhou
azhouad@cse.ust.hk
The Hong Kong University of Science and Technology
China

Heqing Huang
heqhuang@cityu.edu.hk
City University of Hong Kong
China

Hao Ling
hlingab@cse.ust.hk
The Hong Kong University of Science and Technology
China

Charles Zhang
charlesz@cse.ust.hk
The Hong Kong University of Science and Technology
China

Abstract

Static binary bug detection has been a prominent approach for ensuring the security of binaries used in our daily lives. However, the type information lost in binaries prevents the improvement opportunity for a static analyzer to utilize type information to prune away infeasible facts and increase analysis precision. To make binary bug detection more practical with higher precision, in this work, we propose the first hybrid-sensitive type inference, Manta, that combines dataflow analysis with different sensitivities to complement each other and infer precise types for many variables. The inferred types are then used to assist with bug detection by pruning infeasible indirect call targets and data dependencies. Our experiments indicate Manta outperforms prior work by inferring types with 78.7% precision and 97.2% recall. Based on the inferred types, we can prune away 63.9% more infeasible indirect-call targets compared to existing type analysis techniques and perform program slicing on binaries with 61.1% similarity to that on source code. Moreover, Manta has led to 86 new developer-confirmed vulnerabilities in many popular IoT firmware, with 64 CVE/PSV IDs assigned.

ACM Reference Format:
Chengfeng Ye, Yuandao Cai, Anshunkang Zhou, Heqing Huang, Hao Ling, and Charles Zhang. 2024. Manta: Hybrid-Sensitive Type Inference Toward Type-Assisted Bug Detection for Stripped Binaries. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3622781.3674177

*Corresponding author.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0391-1/24/04
https://doi.org/10.1145/3622781.3674177

1 Introduction

Static binary analysis is one of the most powerful techniques for security analysis, which has been widely used in binary-only scenarios, such as detecting vulnerabilities in embedded firmware [15, 64, 81], and industrial control systems [62]. Despite its usefulness, existing binary-level static bug detection tools still suffer from the problem of low precision. Existing studies [52, 54] have revealed that the main reason for this precision gap is the loss of type information in binaries. Type information has been widely utilized to filter infeasible targets in both indirect call analysis [8, 53] and data dependency analysis [11, 25, 54, 66]. However, the types lost during compilation prevent the opportunities to utilize type information to build a better bug-detection tool for binaries. To make type-assisted binary static analysis possible, in this paper, we explore applying type inference to recover variable types in stripped binaries with high precision and recall to assist with static bug detection in binaries.

Problem. Despite binary type inference being a fundamental topic with long-term development, existing techniques still have the following limitations. That is, they either (i) produce an *over-approximated* type inference result or (ii) infer *unknown types* for many variables. As a result, the inferred types cannot effectively be utilized to assist with static binary analysis.

Specifically, on the one hand, a low-precision type inference, such as flow-insensitive or context-insensitive analysis [26, 45, 57], would indiscriminately collect and unify all possible type hints to infer each variable's type. Since the collected type hints could correspond to many variables of different types, the inferred type for each single variable

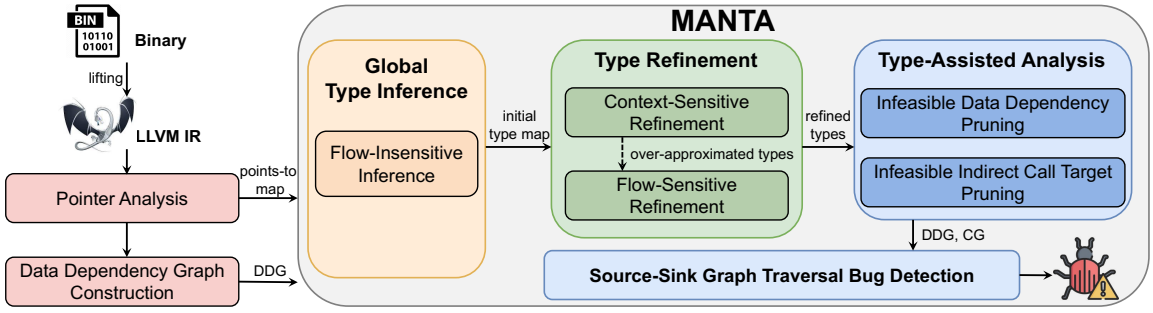
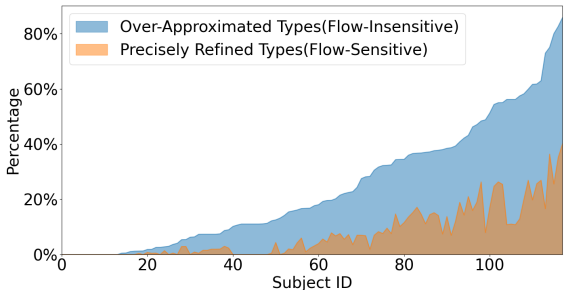


Figure 1. Overall Design of MANTA.

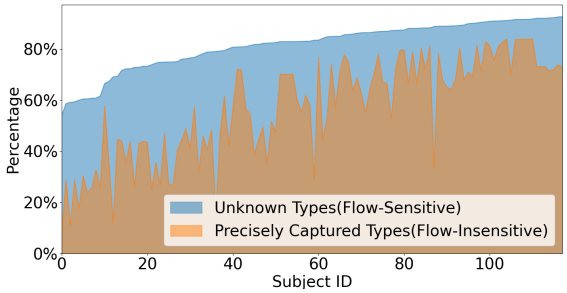
would be *over-approximated*. As represented by the blue region in Figure 2(a), where the profiling data is collected from our experiment on several real-world projects and benchmarks[1], a flow- and context-insensitive analysis would over-approximately infer types for many variables. On the other hand, a high-precision type inference, such as flow-sensitive or context-sensitive analysis [4, 29, 46, 59, 61], would avoid indiscriminately unifying type hints together by propagating type hints along control flow equipped with strong updates, or by propagating type hints along valid calling contexts. However, they cannot infer the type of a variable if no valid type hints are collected, thereby leaving types of many variables to be *unknown*. The blue region in Figure 2(b) reveals a large proportion of variables whose type cannot be inferred by a flow- and context-sensitive analysis.

Observation. Our insight is that a hybrid combination of analyses with different sensitivities can complement each other, allowing for the precise type inference for many variables. On the one hand, a high-precision analysis (e.g., flow-sensitive) could precisely infer types over-approximately inferred by a low-precision analysis. Based on the results of our experiments, the brown region in Figure 2(a) shows the proportion of over-approximated type variables precisely inferred by a flow- and context-sensitive analysis. On the other hand, for variables whose types are inferred as *unknown* by a high-precision analysis, a low-precision analysis could instead capture ignored type hints and precisely infer the variable's type. The brown region in Figure 2(b) shows the distribution of *unknown* type being able to be precisely inferred by a flow- and context-insensitive analysis.

Solution. Based on this observation, we can precisely infer types for more variables by using a high-precision analysis to refine over-approximated types, and using a low-precision analysis to infer types for *type-unknown* variables. To do so, we design the first hybrid-sensitive type inference approach, as shown in Figure 1, that progressively increases precision to infer variable types. Specifically, it starts with a low-precision global flow- and context-insensitive analysis to thoroughly infer types for as many variables as possible. Then, only for the over-approximated types, higher precision



(a) Over-Approximated Types Refined by High Precision Analysis.



(b) Unknown Types Precisely Captured by Low Precision Analysis.

Figure 2. Profiling data on 118 binaries from experiments.

context-sensitive and flow-sensitive analyses are conducted progressively on top of the data dependence graph (DDG) and control flow graph (CFG) to infer more precise type results until the type is precisely resolved as a singleton. In this way, the hybrid-sensitive analysis can infer less over-approximated variable types with staging refinement and be able to infer types for more variables by starting with lower precision analyses to capture type hints thoroughly.

Similar to existing binary type inference methods [26, 45, 57], achieving absolute soundness in inferring all types is challenging for MANTA. Nevertheless, with a high recall rate of 97.2%, MANTA can effectively assist with practical bug detection in stripped binaries from two perspectives. First, a type-based indirect call analysis utilizes the inferred types to validate the type compatibility between arguments at indirect call sites and function parameters to filter infeasible

[1]The used projects and benchmarks are the same as those listed in Table 3

indirect-call targets. Second, type-based data dependency refinement can utilize the inferred types to identify the based pointer at each binary arithmetic instruction, helping with more precise DDG construction. Then, a program slicing-based bug detection technique [67, 72] is performed on DDG to precisely detect a series of vulnerabilities (§5.2).

We have implemented a tool called MANTA, and exhaustively evaluated it against several type inference techniques and binary bug detection tools on several real-world projects and IoT firmware. We make the following contributions:

- *Hybrid-Sensitive Type Inference*: We propose a hybrid-sensitive type inference to infer types for most variables precisely. Experiment shows MANTA infers types with precision by 78.7% and recall by 97.2%, outperforming existing type inference by 32.7% on average.
- *Type-Assisted Static Analysis*: With the assistance of the hybrid-sensitive type inference, MANTA can prune 63.9% more infeasible indirect-call targets than both TYPEARMOR [76] and τ-CFI [55], and perform program slicing on binary for bug detection with 61.1% similarity as on the source code.
- *Many Critical Vulnerabilities*: We have applied MANTA to several IoT firmware samples and detected security bugs with only 22.1% false positive rate, among which 86 are confirmed and fixed by developers while 64 critical vulnerabilities assigned with CVE/PSV IDs.

2 Background and Motivation

We first introduce the limitation of existing type inference in §2.1 and §2.2 and show the advantage of MANTA in §2.3.

2.1 Previous Limitations on Type Inference.

Over-approximated types and unknown types are two major problems encountered by existing type inference.

Over-Approximated Types. Over-approximated types arise from the conflicting type hints captured by the type inference. When a variable is analyzed to be of different types, a principled type inference [26, 45, 57] will merge these types together into an over-approximated result on the lattice. Common reasons for conflicting types include:

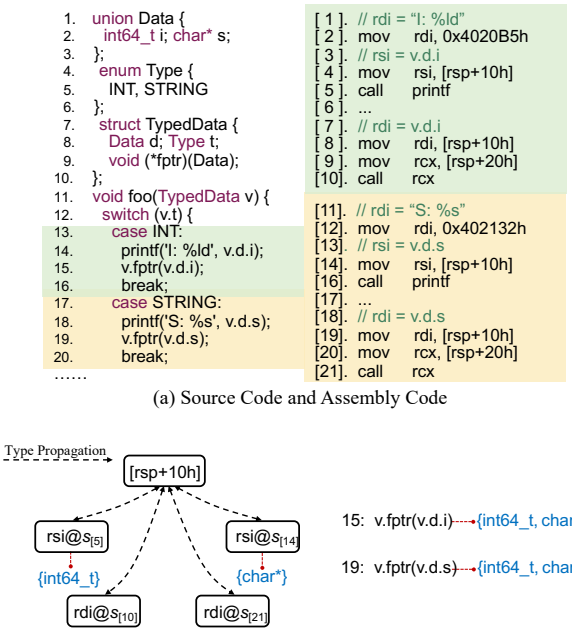
- *Type-Unsafe Idioms.* C and C++ are type-unsafe languages, thus the type of a variable can be explicitly or implicitly converted to other not-compatible types.
- *Union Type.* Union is a language feature where a variable can be declared as multiple types and then instantiated to one of them at different program locations.
- *Stack Recycling.* After compilation, a stack slot could correspond to different variables declared in the same function, which could be of different types.
- *Polymorphic Function.* For a polymorphic function [57], the types of the same arguments or the return value could be different under different calling contexts.

Unknown Types A precise static analysis [29] can avoid conflicting types to mitigate the problem of over-approximated types. For example, a context-sensitive type inference [57] can avoid conflicting types brought by polymorphic functions, as a unique type variable is created for each function argument at each calling context. However, such treatment can introduce side effects for non-polymorphic functions. It is possible that there are no type hints to infer the type of a variable along a calling context, and the context-sensitive analysis prevents the chance to capture type hints from other calling contexts. Consequently, no type hints are captured to infer type of the corresponding variable. Similarly, a flow-sensitive type inference restricts the type propagation direction and incorporates strong updates to infer precise types of variables at different program positions. However, some potential type hints not aligned with the control-flow order could be ignored, such as type hints from the opposite branch, resulting in the same problem.

2.2 Motivating Example

Figure 3(a) shows a source code example and the corresponding assembly code on which flow-insensitive type inference infers over-approximated types. Inside the branch at Lines 13-16, the union type variable `v` is instantiated as int_{64}, and inside the opposite branch at Lines 17-20, the variable is instantiated as *char**. As shown in Figure 3(b), a flow-insensitive type inference [26, 45, 57] can capture two type hints at the two call site of `printf`, and infers that the variable stored in `[rsp+10h]` could be of both *char** and int_{64}. Consequently, as shown in Figure 3(c), the over-approximated type would affect the effectiveness of type-assisted indirect call analysis. Specifically, all the functions with the first parameter of *char** or int_{64} type would be deemed as valid indirect call targets for both the two indirect call sites. However, in fact, the feasible indirect call targets at Line 15 should only accept int_{64} type argument, and the feasible indirect call targets at Line 19 should only accept *char** type argument.

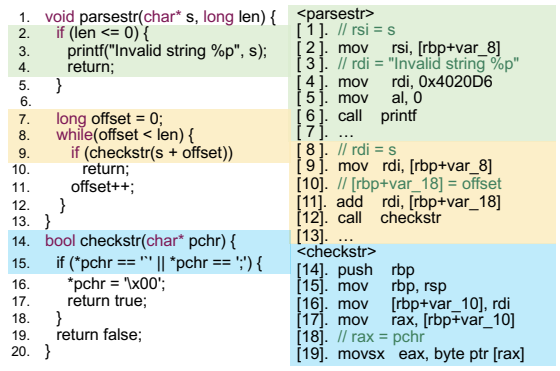
To mitigate the above problem, a flow-sensitive type inference can propagate the two type hints along two branches respectively, and infer precise argument types for the two indirect call sites. However, Figure 4(a) instead shows another example on which a flow-sensitive [4, 29, 46, 59, 61] type inference fails to infer types of variables. The first parameter `s` of the function `parsestr` is used twice inside the function. Inside a security check branch at Lines 2-4, the content of `s` is printed out via `printf`, and then the function returns. Later inside the branch at Lines 7-9, `s` is added by the variable `offset` and passed to the argument `pchr` of the function `checkstr`, in which it is dereferenced. As shown in Figure 4(b), a flow-sensitive type inference captures the type hints at the `printf` (Line [6] in the assembly code) and infers that parameter `s` (`[rbp+var_8]` in the assembly code) is a pointer. However, this type hint cannot be propagated to the opposite branch due to the restriction of control flow order.



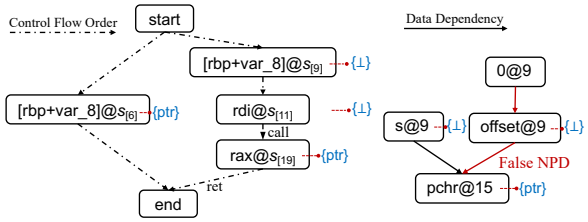
(a) Source Code and Assembly Code

(b) Flow-Insensitive Type Inference (c) Imprecise Indirect Call Analysis

Figure 3. Over-approximated types inferred by flow-insensitive type inference, leading to false call targets.



(a) Source Code and Assembly Code



(b) Flow-Sensitive Type Inference (c) Incorrect Data-Dependency

Figure 4. Unknown types inferred by flow-sensitive type inference, leading to false positive data dependency.

As a result, the type of parameter `s` at Line 9 (`[rbp+var_8]` at Line [9] in the assembly code) is remained unknown. Consequently, as shown in Figure 4(c), a static analyzer cannot distinguish which incoming value of `pchr` is the base pointer, leading to a false NPD value flow path started from a `zero` value to a pointer dereference site of `pchr`.

2.3 Our Approach

In a word, low-precision analysis can infer types potentially missed by high-precision analysis, and high-precision analysis can mitigate the problem of over-approximated types faced by low-precision analysis. As a result, instead of solely relying on any one of them, we design a novel hybrid approach to exploit the benefits of both of them. Specifically, as shown in the workflow graph in Figure 1, a global flow-insensitive type inference is performed first. At this stage, for the example in Figure 4(a), the type of argument `s` is already precisely resolved as *ptr*. However, for the example in Figure 3(a), the two merged types indicate a possibility for further refinement. As a result, context-sensitive and flow-sensitive type refinement with increasing precision is gradually performed on the merged types until it is precisely resolved at the two indirect call sites.

3 Preliminary Definitions

In this section, we give the definitions used in the paper.

Program Abstraction. Figure 5 shows the abstract domain used in our analysis. We utilize binary lifter [38] to translate binary code to LLVM IR, in which binary registers and arguments are translated to SSA value $v \in \mathbb{V}$, and vast binary instruction set for different architectures is mapped LLVM instructions $s \in \mathbb{S}$. Following existing works [47], the global and stack memory region is partitioned into a disjoint set of objects, and the heap object is modeled by allocation-site abstraction. These objects are represented by $o \in \mathbb{O}$. To ensure the analysis scalability, we pre-process the lifted IR to be acyclic by unrolling each loop in the control flow graph (CFG) and the call graph, following the existing bug-finding tools [67, 79].

Points-to Analysis. The points-to-map $\mathbb{P}$ defined in Figure 5 is used to construct the DDG and perform type inference on memory objects. There are plenty of binary points-to analyses, and we follow the state-of-the-art techniques [43, 44] based on the block memory model. The points-to analysis is flow-, field-, and context-sensitive. For scalability consideration, we adopt the bottom-up style compositional technique [78, 79] to avoid reanalyzing the same function from different calling contexts, which has also been used in many well-known bug-finding tools [9, 67, 79].

The points-to analysis involves a few well-identified reasonable unsound choices. Specifically, in the implementation, we follow standard choices to unroll each loop twice, break back edges on the call graph, and collapse fields of an array into a monolithic object when symbolic indexing is encountered. Additionally, following existing works [67, 79], function pointers are not modeled during the points-to analysis. Furthermore, the analysis assumes parameters of a function

$$
\begin{aligned}
\text{Variables} \quad & v \in \mathbb{V} \\
\text{Memory objects} \quad & o \in \mathbb{O} \\
\text{Statements} \quad & s \in \mathbb{S} \\
\text{Points-to map} \quad & \mathbb{P} := \mathbb{V} \cup \mathbb{O} \rightarrow 2^{\mathbb{V} \cup \mathbb{O}} \\
\text{Type map} \quad & \mathbb{F}^{\uparrow}/\mathbb{F}^{\downarrow} := \mathbb{V} \cup \mathbb{O} \rightarrow \mathbb{T}
\end{aligned}
$$

Figure 5. Basic abstract domain.

$$
\begin{aligned}
\text{Type}(\mathbb{T}) \quad &:= \mathbb{T}_{prim} \mid \mathbb{T}_{array} \mid \mathbb{T}_{object} \mid \mathbb{T}_{func} \\
\text{Primary Type}(\mathbb{T}_{prim}) \quad &:= \mathbb{T}_{reg\langle size \rangle} \mid \top \mid \bot \\
\text{Register Type}(\mathbb{T}_{reg}) \quad &:= \mathbb{T}_{num\langle size \rangle} \mid ptr(\mathbb{T}) \\
\text{Numeric Type}(\mathbb{T}_{num<size>}) \quad &:= int_{\langle size \rangle} \mid float \mid double \\
\text{Array Type}(\mathbb{T}_{array}) \quad &:= \mathbb{T} \times \langle length \rangle \\
\text{Object Type}(\mathbb{T}_{obj}) \quad &:= \{\langle offset \rangle_i : \mathbb{T}_i\} \\
\text{Function Type}(\mathbb{T}_{func}) \quad &:= \{arg_i : \mathbb{T}_i\} \rightarrow \mathbb{T} \\
\langle size \rangle \quad &:= \{1, 8, 16, 32, 64\} \\
\langle length \rangle \in \mathbb{N} \quad & \langle offset \rangle \in \mathbb{N}
\end{aligned}
$$

Figure 6. Typing in MANTA

do not alias with each other, easing the effort to build multiple partial transfer functions [78] to model different aliases relationship from the different calling contexts. A further discussion of the implication of these choices, as well as other factors, on the soundness of the system, will be given in § 6.4.

Typing Definition. Figure 6 shows the types supported by MANTA. In general, the type system is similar to that of LLVM, in which we infer pointer type and numeric type with various sizes and precision. Furthermore, the type inference is field-sensitive in inferring object fields and array types. Following existing principled type inference [45], the typing forms a lattice[2], where a type can be a subtype or a parent type of another type. For example, $\mathbb{T}_{num\langle 32 \rangle}$ is the parent type of *float*, denoted by $\mathbb{T}_{num\langle 32 \rangle} >: float$. Symbols $\top$ and $\bot$ denote the upper and lower bound of the type lattice, respectively. For each variable v and memory object o, we maintain its upper bound and lower bound type by two type maps $\mathbb{F}^{\uparrow}(v)$ and $\mathbb{F}^{\downarrow}(v)$.

Next, we give the definition of DDG, on which we can perform type inference and graph-based bug detection.

Definition 1 (Data Dependency Graph). G = (N, E).

- Each of the vertex in N is denoted by $v@s$, indicating that variable v is used or defined at statement s. For example, the instruction $*q = b$ leads to the vertex $b@*q = b$.

[2]The type lattice can be found in Figure 1 inside the supplementary material [3]

Table 1. Rules for global flow-insensitive type inference.

Statement	Typestates Updating
① `COPY:` p = q	`UnifyVarType`(p, q), $\forall\, o_1, o_2 \in \mathbb{P}(p) \cup \mathbb{P}(q)$: `UnifyObjType`($o_1$, o_2)
② `LOAD:` p = *q	$\forall\, o \in \mathbb{P}(q)$: `UnifyVarType`(p, o)
③ `STORE:` *p = q	$\forall\, o \in \mathbb{P}(p)$: `UnifyVarType`(o, q)
④ `TYPE-REVEALING:` (i.e., p reveals as ty)	`UnifyVarType`(p, ty)

- E $\subseteq$ N $\times$ N is the set of directed edges to represent data dependence relations between vertices.

Most data dependencies can be derived from an instruction itself, such as *copy* and *phi* instructions. However, edges between memory dereference sites rely on the points-to analysis. Specifically, the dependency $\langle p@*a = p,\ q@q = *b \rangle$ is constructed if and only if $\exists o \in \mathbb{P}(b),\ p \in \mathbb{P}(o)$, indicating b points to a memory location containing p while it is loaded.

4 Hybrid Sensitive Type Inference

In this section, we describe the global flow-insensitive type inference in §4.1 and the two-stage type refinement in §4.2.

4.1 Global Flow-Insensitive Type Inference

At this stage, a global flow-insensitive type inference is performed to infer types thoroughly. To do so, we apply a unification-based algorithm to unify variable types together.

The type maps are updated during the unification process. $\mathbb{F}^{\uparrow}$ is updated with the join ($\vee$) operator on the type lattice to maintain an upper-bound type, and $\mathbb{F}^{\downarrow}$ is updated with the meet ($\wedge$) operator to maintain a lower-bound type. Prior the analysis, each variable and memory field of $\mathbb{F}^{\uparrow}$ is initialized as $\bot$, and that of $\mathbb{F}^{\downarrow}$ is initialized as $\top$. Typing rules for each kind of instruction are shown in Table 1.

- For value copy instructions ①, including *bitcast*, *phi*, and *call*, `UnifyVarType()` is applied to unify the types of incoming and outgoing values together to update both the two type maps. Additionally, `UnifyObjType` is applied on the pointed-to-by objects to unify types of memory fields sharing the same offset.
- For memory load instructions ②, the types of the variable loaded from the memory field and the types of the field are unified. A similar handling process is performed on memory store instructions ③.
- Type-revealing instructions ④ provide type hints for type inference. Examples include type-known external functions such as `malloc()`, arithmetic calculations, or pointer dereference.

Once the global flow-insensitive type inference finishes, the result of initial type maps is obtained. According to the

type map, each variable is classified into one of the following three categories:

- (**Precise Type Variable**, $\mathbb{V}_P$). The set of variables v whose type is precisely resolved as a singleton, $v \in \mathbb{V}_P : \mathbb{F}^{\uparrow}(v) = \mathbb{F}^{\downarrow}(v)$, since the upper bound and lower bound types are the same.
- (**Over-Approximated Type Variable**, $\mathbb{V}_O$). The set of variables v whose type is over-approximately inferred, $v \in \mathbb{V}_O : \mathbb{F}^{\uparrow}(v) >: \mathbb{F}^{\downarrow}(v)$ and $\mathbb{F}^{\uparrow}(v) \neq \mathbb{F}^{\downarrow}(v)$, since the interval between upper bound and lower bound types could be further narrowed down.
- (**Unknown Type Variable**, $\mathbb{V}_U$). The set of variables v whose type is unknown, $v \in \mathbb{V}_U : \mathbb{F}^{\uparrow}(v) = \bot$ and $\mathbb{F}^{\downarrow}(v) = \top$, since no type hints have been captured to update the variable's type map during the analysis.

For $v \in \mathbb{V}_P$, no further refinement needs to be performed since its type is already precisely resolved, and refinement cannot generate a better result.

Also, no further refinement should be performed on $v \in \mathbb{V}_U$, since even a flow-insensitive type inference cannot capture any type hints to infer its type. Note that for conservative consideration, their upper bound $\mathbb{F}^{\uparrow}(v)$ would be updated to $\top$ and lower bound $\mathbb{F}^{\downarrow}(v)$ would be updated to $\bot$ once the analysis finishes, indicating an *any-type* variable.

Only for $v \in \mathbb{V}_O$, higher precision analysis could further narrow down the type interval to increase the type inference precision. In the next section, we give more details on how to refine the types of these variables.

4.2 Type Refinements

4.2.1 Context-Sensitive Type Refinement. For each $v \in \mathbb{V}_O$, context-sensitive and flow-sensitive analyses will be performed progressively to refine the type map. To achieve this, DDG performs on-demand analysis on these over-approximated type variables.

To infer precise types for variables $v \in \mathbb{V}_O$, we perform context-sensitive graph traversal on DDG to collect more condensed type hints in the sense that:

- Context-sensitivity can be achieved via graph traversal concerning CFL-reachability [65], mitigating over-approximation brought by polymorphic functions.
- Only aliased variables would be searched on the DDG, mitigating the over-approximation brought by merging types of non-aliased variables.

Algorithm 1 shows the pseudocode for the process:

- `CTX_REFINEMENT()` shows the main procedural of the refinement. For each variable $v \in \mathbb{V}_O$, the function first collects its root values by backward traversal on the DDG with `FIND_ROOTS()`. Then all the type hints on the derivatives of the root nodes are collected via `COLLECT_TYPES()`, which will be merged together and used to update the type map of variable v.

Algorithm 1: Context-Sensitive Type Refinement

```
1  Input Data Dependency Graph DDG.
2  Input Over-Approximated Type Variables V_O.
3  Function CTX_REFINEMENT():
4      for v ∈ V_O do
5          types ← ∅
6          for root ∈ FIND_ROOTS(v) do
7              types ← types ∪ COLLECT_TYPES(root)
8          if types ≠ ∅ then
9              F↑(v) ← LUB(types)
10             F↓(v) ← GLB(types)
11 Function FIND_ROOTS(v):
12     root_sets ← ∅
13     for p in DDG.parents(v) do
14         ctx_stack.insert(context(p, v))
15         if ctx_stack.valid() then
16             root_sets ← root_set ∪ FIND_ROOTS(p)
17         ctx_stack.pop()
18     if root_sets == ∅ then
19         root_sets ← v
20     return root_sets
21 Function COLLECT_TYPES(v):
22     type_sets ← type_annotations(v)
23     for c in DDG.childs(v) do
24         cfl_stack.insert(context(p, v))
25         if ctx_stack.valid() then
26             type_sets ← type_sets ∪
                COLLECT_TYPES(p)
27         cfl_stack.pop()
28     return type_sets
```

- `FIND_ROOTS()` shows the procedure to find the root nodes via backward traversal on the DDG. One key point is that *ctx_stack* maintains the calling context while traversing the DDG, and unreachable call contexts would be rejected to achieve context sensitivity. Note that since recursive cycles have been removed from the program during pre-processing, calling contexts can be tracked via pushing and popping from a stack, without risk of non-termination. Additionally, when MANTA encounters a binary instruction such as *add* or *sub* during traversal, it would turn to resolve the type of operands first and performs feasibility checking (discussed in §5.2) to determine the correct searching direction.
- `COLLECT_TYPES()` shows the procedural to collect type hints of a root node. At a high level, a forward DDG traversal with CFL-reachability validation is performed

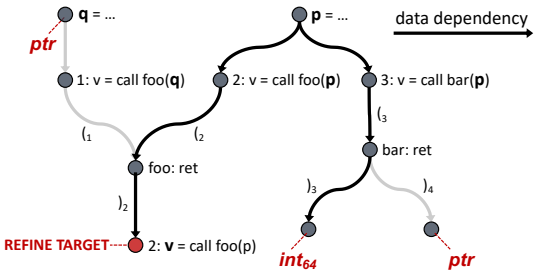


Figure 7. Context-Sensitive Type Refinement

on the root node, and all the type annotations on the traversed nodes will be collected and returned.

Example 4.1. Figure 7 shows an example of context-sensitive type refinement on v. To refine the type, a context-sensitive backward search first detects the root node p. Next, all valid type hints are collected through a forward CFL-reachable traversal on p, and only the type hint for int_{64} is collected. The two type annotations of ptr are not collected due to CFL-unreachable paths, resulting in both $\mathbb{F}^{\uparrow}(v)$ and $\mathbb{F}^{\downarrow}(v)$ being precisely resolved as int_{64}.

4.2.2 Flow-Sensitive Type Refinement.

For a variable whose type is still over-approximated, we further perform flow-sensitive type refinement to refine its type. Given an over-approximated type variable $v \in \mathbb{V}_O$, the flow-sensitive refinement is performed in two ways. First, only reachable type hints following the control-flow order are collected to reveal variables' types. Second, the def- and each use-site s of variable v, denoted by $v@s$, would be treated as a distinct variable to be inferred its type. In our formalism, the type of v at location s is represented by $\mathbb{F}^{\uparrow}(v@s)$ and $\mathbb{F}^{\downarrow}(v@s)$, based on an extended notation of the type map defined in Figure 5. For variable $v \in \mathbb{V}_U \cup \mathbb{V}_P$, at each of its use site s we have $\mathbb{F}^{\uparrow}/\mathbb{F}^{\downarrow}(v) == \mathbb{F}^{\uparrow}/\mathbb{F}^{\downarrow}(v@s)$, since flow-sensitive refinement is not required to be performed.

Algorithm 2 shows the pseudocode for flow-sensitive type refinement:

- `FLOW_REFINEMENT()` shows the main procedural of the refinement. Similarly, for each $v \in \mathbb{V}_O$, the procedural invokes `FIND_ROOTS()` to collect its root values, which can be used to check the alias relationship. Next, at def- and each use-site s of v, we collect type hints on aliases of v reachable to s via `REACHABLE_TYPES()`. The collected type hints would be merged to represent the inferred type of $v@s$.
- `REACHABLE_TYPES()` performs backward searching on *CFG* to collect type annotations on aliases of v reachable to s. Specifically, if the currently traversed statement is a type-revealing site of an alias of v, the corresponding revealed type would be collected. The searching stops when a type annotation is met during the traversal such that strong update is applied.

Algorithm 2: Flow-Sensitive Type Refinement

```
1  Input Data Dependency Graph DDG.
2  Input Control Flow Graph CFG.
3  Input Over-Approximated Type Variables V_O.
4  Function FLOW_REFINEMENT():
5      for v ∈ V_O do
6          v_roots ← FIND_ROOTS(v)
7          for s ∈ get_users(v) ∪ {v} do
8              types ← REACHABLE_TYPES(s, v_roots)
9              if types ≠ ∅ then
10                 F↑(v@s) ← LUB(types)
11                 F↓(v@s) ← GLB(types)
12 Function REACHABLE_TYPES(s, roots):
13     for v ∈ get_oprands(s) ∪ {s} do
14         if FIND_ROOTS(v) ∩ roots ≠ ∅ then
15             if type_annotation(v@s) ≠ ∅ then
16                 return type_annotation(v@s)
17     types ← ∅
18     for p in CFG.parents(s) do
19         ctx_stack.insert(context(p, s))
20         if ctx_stack.valid() then
21             types ← types ∪
                   REACHABLE_TYPES(p, roots)
22         ctx_stack.pop()
23     return types
```

The flow-sensitive refinement can effectively handle the over-approximation introduced by type casting or type instantiation of variables at different use sites, while also mitigating some compiler optimizations, such as stack recycling, where multiple variables of different types can be stored in the same stack location at different program points.

Example 4.2. Figure 8 shows the flow-sensitive type refinement on the example in Figure 3. To infer the type of $v.d$ used as function arguments at two indirect call sites, MANTA performs backward searching on CFG from these two sites, respectively. For the use site at Line 15, one type annotation of int_{64} at Line 14 is collected; for the use site at Line 19, one type annotation of int_{64} at Line 18 is collected. As a result, the type of $v.d$ can be precisely inferred as int_{64} and $ptr(int_8)$ at two call sites, respectively.

5 Type-Assisted Static Analysis

In this section, we describe the type-assisted static analysis clients in detail, including type-based indirect call analysis in §5.1 and type-based data-dependency pruning in §5.2.

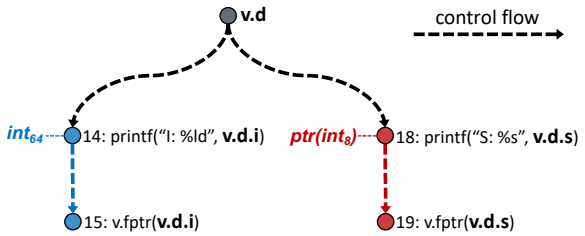


Figure 8. Flow-Sensitive Type Refinement

5.1 Type-Based Indirect Call Analysis

At a high level, inferred types of arguments and return values at each indirect call and address-taken function are checked to validate the calling feasibility. Differing from existing works [55, 76] in which only argument number and width are recovered for compatibility checking, richer types in MANTA can prune away more infeasible indirect call targets effectively.

We validate the indirect call feasibility from the following perspectives:

- The number of the arguments at the call site should be greater than that of the function definition.
- For each actual argument arg_i at indirect call site s and parameter par_i of function f, their types should satisfy that $\mathbb{F}^{\uparrow}(arg_i@s) >: \mathbb{F}^{\downarrow}(par_i@entry_f)$.
- For actual return value ret at call site and formal return value ret_f at function definition, their types should satisfy that $\mathbb{F}^{\uparrow}(ret_f@exit_f) >: \mathbb{F}^{\downarrow}(ret@s)$.

For pointer type and memory type, type comparison is performed on each field recursively.

Example 5.1. Back to figure 8, the type of argument $v.d.s$ of the indirect call site at Line 19 is precisely inferred as $ptr(int\_8)$. As a result, only functions with at most one argument whose lower bound type is at most $ptr(int\_8)$ can be feasible indirect call targets.

5.2 Infeasible Data Dependency Pruning

The inferred types can also help us prune away infeasible data dependency regarding pointer arithmetic calculation, in case these data dependency relationships lead to false positives while performing program slicing on DDG. Table 2 shows the detailed rules for the pruning.

- For *add* instruction, if the result is of *ptr* type, then this instruction should be pointer arithmetic in which a base pointer is added by an offset of *numeric* type. In such a case, we prune away the data dependency from the numeric operand to the result pointer since the numeric operand is not an alias of the result pointer.
- For *sub* instruction, if the result is of *numeric* type and the operand is of *ptr* type, then this instruction should calculate the offset between two pointers. In such case, we prune away the dependency from *ptr* type operand to *numeric* type result since the calculation result is not the alias of the pointer operand anymore. Similarly, if the result is of *ptr* type, then the dependency from the second operand to return is pruned.

Table 2. Rules for pruning infeasible data dependency. The second column denotes the rule, in which $\texttt{TY}(v) = ty$ is the abbreviation of "$\mathbb{F}^{\uparrow}(v) = \mathbb{F}^{\downarrow}(v) = ty$". The third column denotes the pruned data dependency, in which $v \longrightarrow r$ represents the pruned data dependency from v to r.

Opcode	Rules	Infeasible Dep
`s: R = ADD OP1, OP2`	$\texttt{TY(R@s)} = ptr \wedge \texttt{TY(OP1@s)} = \mathbb{T}_{num}$	OP1 ⟶ R
`s: R = ADD OP1, OP2`	$\texttt{TY(R@s)} = ptr \wedge \texttt{TY(OP2@s)} = \mathbb{T}_{num}$	OP2 ⟶ R
`s: R = SUB OP1, OP2`	$\texttt{TY(R@s)} = \mathbb{T}_{num} \wedge \texttt{TY(OP1@s)} = ptr$	OP1 ⟶ R
`s: R = SUB OP1, OP2`	$\texttt{TY(R@s)} = \mathbb{T}_{num} \wedge \texttt{TY(OP2@s)} = ptr$	OP2 ⟶ R
`s: R = SUB OP1, OP2`	$\texttt{TY(R@s)} = ptr$	OP2 ⟶ R

Example 5.2. Back to the example in Figure 4(c), there is a false positive NPD value flow path starting from constant `0` at Line 9 to `*pchr` at Line 15. If the type inference can infer that the type of `offset` at line 9 is of *numeric* type, then the data dependency edge from `offset` to `*pchr` can be pruned away. As a result, the false positive NPD will not be detected.

5.3 Source-Sink DDG Traversal Bug Detection

Following past works [72, 75], we model the bug detection as program slicing over DDG with path-feasibility validation. For example, an NPD vulnerability can be detected by capturing a feasible value flow path from which a NULL value can flow to a pointer dereference site.

By describing the specification of sources and sinks, a series of vulnerabilities can be detected. Specifically, in this paper, we set up example checkers for five representative security bugs for evaluation purposes, including Null Pointer Dereference (NPD), Return Stack Address (RSA), Use After Free (UAF), OS Command Injection (CMI) and Buffer Overflow (BOF). (The details of these specifications are shown in Table 4 inside the supplementary material [3]). Besides, users of MANTA can easily implement a new bug checker by specifying the sources and sinks of the vulnerabilities to detect.

6 Evaluation

This section evaluates the following research questions:

- **RQ1:** How effective is the hybrid-sensitive type inference of MANTA compared with existing echniques?
- **RQ2:** How effective are the inferred types of MANTA assisting with static analysis compared with existing type inference techniques?
- **RQ3:** How effective is MANTA as a static binary bug-finding tool compared to others?

Benchmark. The benchmarks for type inference evaluation contain 14 large-scale open-source projects, and the

Table 3. Type inference precision and recall on variables in 14 large-scale open source projects and coreutils benchmarks.

Project	KLoC	#Vars	Dirty [17]		Ghidra [5]		RetDec [38]		Retypd [57]		Manta							
											FI		FS		FI + FS		FI + CS + FS	
			%Prec	%Recl	%Prec	%Recl	%Prec	%Recl	%Prec	%Recl	%Prec	%Recl	%Prec	%Recl	%Prec	%Recl	%Prec	%Recl
vsftpd	16	765	68.0	89.2	33.9	59.7	36.5	36.5	25.4	89.1	13.3	97.9	26.8	98.6	38.6	96.6	77.1	94.2
libuv	36	1,212	71.2	97.6	31.6	51.1	39.6	39.6	29.4	86.3	41.9	98.9	21.6	99.7	56.1	98.9	85.0	98.9
memcached	48	1,033	28.3	90.6	19.1	34.8	49.4	49.4	3.2	98.6	37.1	96.2	32.0	98.8	58.8	95.9	83.2	95.5
lighttpd	89	2,454	60.1	88.8	36.1	54.3	39.6	39.6	△		17.9	99.5	36.6	99.0	49.4	98.5	93.2	98.0
tmux	110	3,607	63.1	87.6	33.9	59.0	36.3	36.3			25.1	99.4	27.7	98.7	46.7	98.1	82.6	96.6
coreutils	115	35,036	69.8	85.5	31.8	73.4	55.5	55.5	25.9	88.4	60.4	98.7	18.9	99.0	70.2	98.0	84.0	97.7
openssh	119	3,928	61.8	88.3	36.9	68.7	40.7	40.7	31.6	88.3	23.1	99.3	27.2	99.7	44.0	99.0	85.8	98.3
wolfSSL	122	4,634	61.6	87.4	22.6	66.8	40.9	40.9	18.4	88.9	18.7	98.4	30.7	98.0	45.3	96.7	77.7	95.4
redis	179	6,947	55.6	82.1	32.2	56.6	32.9	32.9			28.2	97.5	23.4	98.7	48.0	96.6	78.6	95.7
libicu	317	12,251	52.0	91.5	41.0	68.6	40.6	40.6			27.3	99.3	25.5	99.3	48.3	98.7	86.1	98.3
vim	416	13,891	‡		34.2	54.4	47.1	47.1			25.1	99.4	43.6	98.9	60.8	98.4	85.5	97.8
python	560	11,178			45.7	67.6	24.7	24.7	△		17.5	99.7	15.9	99.4	32.3	99.1	71.1	98.1
wrk	594	14,225	78.5	90.8	67.2	78.5	27.2	27.2			25.1	98.9	13.0	99.4	36.7	98.4	74.8	96.3
ffmpeg	1,213	68,576	72.4	85.1	22.8	60.3	41.8	41.8			45.2	97.7	16.4	99.4	56.5	97.3	72.9	96.7
php	1,358	20,007	43.7	87.4	30.9	59.7	31.2	31.2			28.9	98.6	22.8	98.9	47.1	97.8	77.2	97.2
Total.	**5,177**	**223,256**	**63.7%**	**86.9%**	**32.2%**	**64.0%**	**41.0%**	**41.0%**	**25.2%**	**88.6%**	**35.9%**	**98.5%**	**22.3%**	**99.2%**	**53.1%**	**97.9%**	**78.7%**	**97.2%**

△ denotes the type inference cannot finish analysis in 72 hours.
‡ denotes the type inference crashes.

coreutils benchmarks include 104 separate binaries. These projects range from tens of thousands to millions of lines of code, covering a broad spectrum of applications such as servers, databases, and widely used third-party libraries. We compiled these projects into binaries using the building systems' default optimization settings, including -O2, -O3, -Ofast, to conform to the real-world scenario. To demonstrate Manta's effectiveness in real-world binary bug detection, we also selected 9 IoT firmware samples from famous vendors and performed bug detection on them. All experiments were conducted on a server with two 20-core Intel(R) Xeon CPU@2.20GHz CPU and 256GB physical memory running Ubuntu-20.04.

6.1 Comparison with Existing Type Inference

Metrics. We evaluate the ability to infer first-layer types of function parameters and measure the quality by precision (P) and recall (R). We measure precision by the proportion of variables whose type is correctly inferred and measure recall by the proportion of variables whose type is captured by the type inference. For example, suppose a type inference concludes that a type is *unknown* (considered as any types in the analysis) or produces a range including the actual type; the case will be counted toward the recall contribution since the inferred type results include the actual type. To obtain ground truth variable types for evaluation, we parse the DWARF debugging information to get the defined types of variables in the source code. As addressed in ReTypd's [57] evaluation, in a few cases, source types could be less precise than what a type inference tool can infer due to type-unsafe usages. For instance, a variable may be defined as *uint64_t* but actually used as a pointer. Following the existing evaluation convention, we did not specifically address these cases and evaluated all the tools with the same metrics to ensure fairness in the evaluation.

We compare Manta with four existing type inference tools: Dirty [17], Ghidra [5], RetDec [38] and Retypd [57]. Furthermore, we set four comparison groups for Manta. Manta-FI only performs the imprecise flow-insensitive type inference, Manta-FS only performs flow-sensitive analysis, Manta-FI+FS combines them together in a staging approach, and Manta-FI+CS+FS performs the full-stage analysis.

Table 3 shows the type inference precision and recall on benchmarks. Among all the comparison groups, Manta-FI+CS+FS achieves the best result where the precision is **78.7%** and recall is **97.2%**, outperforming Dirty, Ghidra, RetDec and Retypd, whose precision and recall is **63.7%/86.9%**, **32.2%/64.0%**, **41.0%/41.0%** and **25.2%/88.6%** respectively.

Analysis of Other Tools. Dirty is the state-of-the-art data-driven type inference technique that achieves a good result. It is powerful and can predict variable and structure names, even though our downstream static analysis application does not request such information. However, since these data-driven approaches guess types, they cannot have high recall as Manta and cannot achieve high precision as the prediction could be incorrect. Ghidra is a famous decompiler equipped with type inference. It performs a heuristic rule-based analysis by modeling some access patterns and only performs regional type propagation. Thus, it cannot achieve a satisfactory inference result. During the manual inspection, we also noticed that many variables are inferred as *undefined* when there are no hints collected to infer the type. RetDec is a binary lifter equipped with a type inference technique similar to Ghidra, but the difference is that it does not produce *unknown* type since its output should be a valid LLVM IR in which all values should have type. As a result, it will mark the value whose type cannot be inferred as *i32*; such treatment introduces low recall as lots

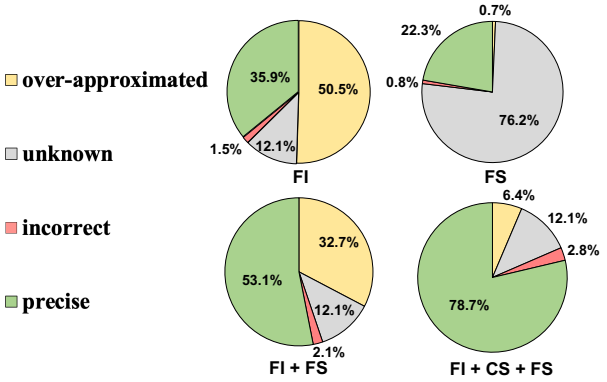


Figure 9. The proportion of inferred types result generated by a combination of different analysis sensitivity.

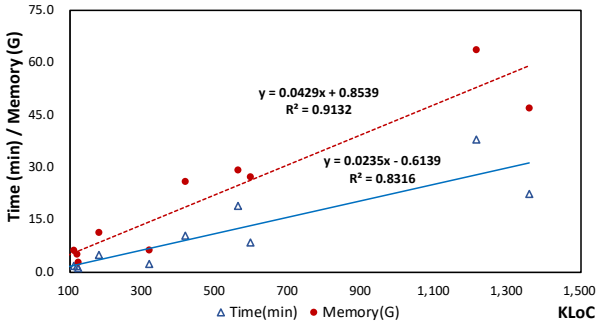


Figure 10. Type inference performance based on runtime data. The x-axis stands for the number of lines in a project (KLoC) and the y-axis stands for the time and memory cost.

of pointer type variables are inferred as integer type. RETYPD is a principled type inference based on a subtyping system and is implemented as an open-source plugin on top of GHIDRA. However, its core is a constraint-solving engine performing transitive closure analysis with $O(N^3)$ time complexity, which is inefficient when analyzing large binaries. Our experiment shows it can only finish analysis on a few large-scale projects in 72 hours.

Ablation Analysis. The standalone MANTA-FI and MANTA-FS cannot achieve high precision. We investigate the deep reason behind this by analyzing the distribution of inferred types. As shown in Figure 9, 50.5% of types are over-approximately analyzed by MANTA-FI, while 76.2% *unknown* types cannot be inferred by MANTA-FS. By combining both, MANTA-FI+FS can infer types for a large portion of *unknown* types variable and refine many over-approximated types, contributing to higher precision. However, some over-approximated types will be directly refined as *unknown* in the flow-sensitive refinement stage. To mitigate the problem, MANTA-FI+CS+FS utilizes a flow-insensitive but context-sensitive intermediate stage to resolve these over-approximated types in case they are directly inferred as *unknown* due to ignoring all the potential type hints. As a result, MANTA-FI+CS+FS achieves the highest precision. The result also reveals a slight side effect; the higher precision comes with a few incorrect inferred types, leading to lower recall. The core reason behind the phenomenon is that some typing rules are not always correct due to the type-unsafe usage in the C/C++ language. For example, a typical case is that some pointer-type variables could be compared with a special integer indicating an error value (e.g., -1). Since the typing rules inside MANTA decide that two compared variables should have the same types, it will incorrectly infer the pointer as an integer in such a case.

Scalability. MANTA can scale to large binaries with acceptable time and memory resources. For example, even on large projects like `FFmpeg` with a million lines of code, MANTA can finish the type inference in 38 minutes with 64G memory. Figure 10 further gives the fitting curves over the data collected from evaluation, showing that the performance almost increases linearly as the project size increases.

6.2 Downstream Evaluation on Static Analysis

In this section, we evaluate the effectiveness of the inferred types in assisting with two downstream static analysis tasks. The first task is to utilize the inferred type to resolve indirect call targets, and the second task is to refine the DDG and better help with program slicing for bug detection.

6.2.1 Type-Assisted Indirect Call Analysis.

We evaluated type-based indirect call analysis on the 14 open-source projects used in previous experiments. Furthermore, except for the above type inference tools, we further compared with two existing type-based binary indirect call analysis TYPEARMOR [76] and τ-CFI [55]. TypeArmor uses the number of arguments to prune away infeasible targets, and τ-CFI further utilizes the width of arguments. Since the type information has already been included in MANTA, we compared it with them based on our implementation.

Metrics. To obtain the ground truth data, we utilized the type in the source code to perform the type-based indirect call analysis [8]. The indirect call targets pruned away by the source-level analysis are considered infeasible, and the remaining are considered feasible. To match the results between source- and binary-level analysis, we kept *.debug_line* sections in binary to map each indirect call instruction to its source file position and to obtain the names of address-taken functions. Precision and recall can be calculated with a standard formula, where higher precision indicates more pruning on infeasible targets, and higher recall means less incorrect pruning on feasible targets. Note that even though some feasible targets generated by source-level type-based indirect call analysis contain false positives, a binary-level type-based approach still should not refine more targets than the source-level counterparts. Besides, following existing works, we analyzed the Average Indirect Call Target (AICT) to measure the effectiveness of the pruning.

Table 4. Type-based indirect call analysis Average Indirect Call Targets (#AICT) and precision (P) assisted by type inference.

Project	# AT	Source	Dirty [17]	Ghidra [5]	RetDec [38]	Retypd [57]	TypeArmor [76]	τ-CFI [55]	MANTA			
									FI	FS	FI + FS	FI + CS + FS
		#AICT	#AICT (P)	#AICT (P)	#AICT (P)	#AICT (P)	#AICT (P)	#AICT (P)	#AICT (P)	#AICT (P)	#AICT (P)	#AICT (P)
1. vsftpd	29	4	23.4 (21.4%)	24.0 (18.6%)	19.0 (33.4%)	24.5 (16.8%)	26.4(10.3%)	25.9(12.3%)	25.6(13.2%)	25.9(12.3%)	25.0(15.7%)	22.8(24.5%)
2. libuv	87	11	58.5 (36.3%)	35.6 (59.0%)	50.4 (45.9%)	52.0 (43.1%)	74.5(9.2%)	73.4(10.8%)	70.1(15.6%)	70.1(15.6%)	65.2(22.6%)	63.7(24.7%)
3. memcached	64	7	50.2 (24.0%)	27.0 (42.5%)	11.7 (46.2%)	64.0 (0.0%)	60.3(6.5%)	60.0(7.0%)	46.3(30.9%)	60.0(7.0%)	44.3(34.5%)	43.0(36.8%)
4. lighttpd	103	18	72.8 (32.5%)	43.6 (52.6%)	45.9 (57.3%)	△	80.5(24.7%)	78.3(27.3%)	77.6(28.1%)	78.2(27.3%)	76.5(29.4%)	73.3(33.2%)
5. tmux	610	18	348.7 (30.8%)	255.6 (37.1%)	301.5 (37.1%)		497.7(18.8%)	480.2(21.7%)	479.5(21.8%)	480.0(21.7%)	473.7(22.8%)	459.6(25.2%)
6. openssh	177	31	141.0 (20.9%)	81.7 (45.2%)	36.5 (56.0%)	112.3 (38.0%)	127.1(31.3%)	122.8(34.4%)	120.1(36.3%)	122.5(34.6%)	115.5(39.5%)	79.5(65.2%)
7. wolfssl	13	3	7.0 (38.5%)	8.8 (28.5%)	5.8 (35.5%)	6.4 (36.3%)	10.2(28.2%)	9.6(34.3%)	9.5(35.4%)	9.6(34.3%)	9.2(38.2%)	8.9(40.7%)
8. redis	1058	43	657.4 (33.4%)	540.0 (31.8%)	557.9 (38.3%)		856.3(19.3%)	845.9(20.2%)	825.6(22.2%)	844.4(20.4%)	806.6(24.0%)	766.4(28.0%)
9. libicu	1281	53	1152.0 (10.3%)	1239.1 (3.3%)	517.5 (55.6%)		992.7(23.4%)	938.4(27.8%)	909.3(30.2%)	931.8(28.3%)	875.9(32.9%)	807.7(38.4%)
10. vim	1251	32	‡	307.0 (36.6%)	361.6 (36.0%)	△	798.8(36.9%)	767.8(39.4%)	745.5(41.2%)	763.5(39.7%)	704.9(44.6%)	600.5(53.2%)
11. python	3128	336		989.8 (40.8%)	886.0 (37.9%)		2553.0(20.2%)	2549.6(20.3%)	2541.7(20.6%)	2537.9(20.7%)	2514.0(21.5%)	2398.2(25.5%)
12. wrk	2069	94	1126.9 (44.7%)	733.7 (52.7%)	723.4 (49.2%)		1600.7(23.5%)	1566.3(25.2%)	1547.9(26.1%)	1549.6(26.0%)	1471.2(30.0%)	1336.7(36.8%)
13. ffmpeg	9059	393	8346.6 (8.1%)	5021.8 (33.4%)	3174.8 (47.0%)		7357.8(19.5%)	7122.0(21.9%)	5899.8(35.9%)	7108.4(22.1%)	5512.5(40.3%)	5044.1(45.5%)
14. php	4331	136	3313.9 (20.7%)	2171.4 (30.8%)	2382.1 (39.2%)		3558.2(17.2%)	3522.9(18.0%)	3473.4(19.2%)	3520.0(18.1%)	3431.7(20.2%)	3299.2(23.3%)
Geomean.	**439.6**	**31.9**	**242.1 (24.1%)**	**208.8 (31.6%)**	**169.8 (43.2%)**	**35.8 (31.6%)**	**351.9 (18.8%)**	**342.6 (20.8%)**	**326.8 (25.8%)**	**340.6 (21.6%)**	**315.5 (28.5%)**	**289.8 (34.1%)**

△ denotes the type inference cannot finish analysis in 72 hours.
‡ denotes the type inference crashes.

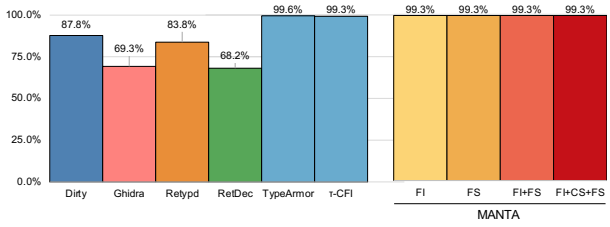


Figure 11. Recall of type-based indirect call analysis.

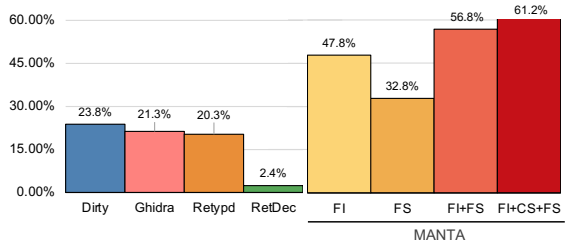


Figure 12. F1 score of source-sink pair program slicing.

Precision. Table 4 shows the indirect call analysis result. In summary, MANTA-FI+CS+FS can prune away much more infeasible indirect call targets than TYPEARMOR and τ-CFI (**34.1% vs 18.8%/20.8%**), indicating its effectiveness. Compared with other type inference tools, MANTA still shows an advantage, benefiting from the high precision of MANTA's inferred types. Despite RETDEC can help with pruning more indirect call targets, we indicate that such precision comes with very low recall, which will be shown below.

Recall. Figure 11 shows the geometric mean of indirect call analysis recall assisted with different type inference. All the ablation groups of MANTA, TYPEARMOR, and τ-CFI exhibit very high recall (99.3%-99.6%), while other type inference tools are relatively low in recall (68.2%-87.8%). As shown by the evaluation, the recall of the type-based indirect call analysis is directly affected by the recall of the type inference, as a type inference tool with more incorrect inferred types would lead to more incorrect pruning of indirect call targets. For example, when assisted with RETDEC with 41.0% recall in type inference, the recall of indirect call analysis is only 68.2%, meaning that a substantial amount of feasible indirect call targets are incorrectly pruned away.

6.2.2 Infeasible Data Dependency Pruning.

To evaluate how the refined DDG can benefit bug detection, we perform program slicing on five security vulnerability types on the refined DDG with different type inference approaches. We measure the similarity between the slicing results generated by Pinpoint on source code.

Metrics. We regard each sliced source-sink pair as a unit and take the result from Pinpoint on source code as ground truth. Then, the F1 score is calculated to measure the similarity between the source-sink pair detected on binary and detected by Pinpoint on source code. Similar to the evaluation of indirect call analysis, *debug_line* section in binary is kept to perform matching between compiled and lifted IR for evaluation purposes.

Result. Figure 12 shows the comparison result on F1 score between different tools[3]. MANTA achieves the highest f1 score by 61.2%. Since the precision of inferred type is lower for other ablation groups, the sliced source-sink pairs contain more false positives as more infeasible data dependency cannot be pruned away. Other type inference has lower f1 scores, ranging from 2.4% to 23.8%. The lower f1 score is due to the lower recall of the type inference, resulting in some real source-sink pairs being incorrectly pruned away. For example, RETDEC infers many pointer-type variables into integer type, leading to incorrect data-dependency pruning.

6.3 Effectiveness of Real-World Bug Detection

To evaluate MANTA's ability to detect bugs in real-world binaries, we compared it with three existing binary bug-finding tools, CWE_CHECKER [1], ARBITER [75] and SATC [15], and manually checked the bug reports generated on nine IoT firmware samples. If one bug report contains more than 100

[3]The detailed evaluation data is shown in Table 2 inside the supplementary material [3]

Table 5. Comparison among bug-finding tools in false positives (#FP), reports (#R), and time in seconds (Time).

Model	Arbiter			cwe_checker			SaTC			Manta			Manta-Notype		
	#FP	#R	Time	#FP	#R	Time	#FP	#R	Time	#FP	#R	Time	#FP	#R	Time
Netgear SXR80	NA	NA	NA	64	68	560	100%†	146	81	14	43	1160	34	64	1758
Zyxel NR7101	0	0	0	20	30	126	0	0	20	5	19	190	12	26	177
Tenda A15	NA	NA	NA	NA	NA	NA	92%†	152	167	6	21	867	13	29	1158
TRENDNet TEW-755AP	NA	NA	NA	64%†	332	278	88%†	143	767	10%†	136	123	26%†	170	211
ASUS RT-AX56U	NA	NA	NA	14	17	190	19	19	337	4	19	175	15	30	277
TOTOLink LR350	0	0	0	4	10	98	97%†	441	132	2	15	63	4	17	66
TOTOLink NR1800X	0	0	0	3	8	112	97%†	346	148	4	28	67	32	56	81
TP-Link WR940N	NA	NA	NA	NA	NA	NA	100%†	661	644	38	99	1760	73%†	236	2825
H3C MagicR200	NA	NA	NA	NA	NA	NA	100%†	146	158	3	8	1053	17	22	1504
FPR	NA		–	72.3%		–	97.4%		–	23.1%		–	52.8%		–

† means we compute the FPR based on 100 randomly-selected reports; NA means the analyzer crashes on the firmware sample.

bugs, we randomly choose 100 to review to ease the manual effort. Furthermore, to validate the effectiveness of the type-assisted static analysis, we add an ablation group Manta-Notype where type compatibility checking is disabled.

Precision. As shown in Table 5, the FPR of cwe_checker, SaTC, Manta, and Manta-Notype is 72.3%, 97.4%, 23.1%, and 52.8%, respectively. In our experiments, Arbiter could not produce any bugs in these benchmarks. Upon profiling it, we found that its under-constrained symbolic execution stage pruned away all the bugs, including some true positives detected by Manta. In conclusion, the comparison indicates that Manta is the most effective in terms of precision and utility. Compared with Manta-Notype, type constraints can significantly reduce false reports by 73.9%, demonstrating the necessity and effectiveness of our type inference.

Comparison with Other Tools. While manually studying the bug reports of other tools, we have the following observations. First, these tools do not utilize type information, so they have higher FPR or limitations in finding certain bugs. For example, some false positives reported by SaTC are due to a tainted string being converted into an integer before reaching *system* function. Thus, attackers cannot directly control the command of *system*. Another example is that cwe_checker's *Missing Null Check* detector cannot detect the case where a null pointer originates from a constant zero value in the binary since it cannot decide whether the zero is an integer or a null pointer.

Comparison with Manta-Notype. The type-assisted analysis results in more precise indirect call analysis and DDG, thus helping Manta detect fewer false positives than Manta-Notype (**23.1% vs 52.8%**). We also checked whether the pruning would result in false negatives. Three of the 216 true bugs detected by Manta-Notype were incorrectly pruned away due to incorrect inferred types. Given that developers typically cannot bear excessive false positives in practice [12], such a small sacrifice is worthwhile for building better static bug detection tools on binaries. Finally, we made an interesting observation while inspecting the time consumption of Manta-Notype and Manta. Despite the additional time costs of type inference, the overall analysis time is reduced. The phenomenon is because inferred types help stop program slicing on incorrect program paths, avoiding unnecessary resource consumption.

Vendor-Confirmed Bugs and Assigned CVEs. Based on our early experiences, directly reporting bugs without a proof of concept (PoC) often results in them being overlooked by developers. This is because the firmware is provided as commercial off-the-shelf binaries without the corresponding source code, making communication with developers about the bugs difficult. To facilitate communication, one of the authors spent a month reverse-engineering the firmware binaries and successfully developing PoCs to trigger a significant proportion of these bugs. However, a few bugs are deeply entwined within complex code logic, presenting challenges for manual reproduction. Therefore, we chose not to pursue further analysis of these bugs' exploitability. At the time of writing, 86 bugs were confirmed with 64 CVE or PSV IDs assigned due to their high-security impacts [4]. The confirmation of bugs spans a broad spectrum of issues, and more excitingly, we have received official acknowledgment and bug bounty rewards from vendors such as ASUS, Zyxel, TRENDNet, and Netgear. This result demonstrates the practicability of Manta.

6.4 Discussion

Soundness. Despite achieving a high recall rate of 97.2% in the evaluation, Manta's type inference still has limitations in inferring all variable types, which is also a common issue suffered by existing binary type inference methods [26, 45, 57]. The little loss in recall can be attributed to two main factors.

First, the modeled typing rules cannot always reflect the correct types of variables due to type-unsafe usage and compiler optimization. For example, it is a common programming

[4]The confirmed bugs list is shown in Table 3 in the supplementary material [3]

idiom to typecast pointer-type variables and compare them with specified constant integers to check for error conditions. Similarly, compiler optimization may introduce bit-level operations on pointer-type variables to ensure proper address alignment. These special cases would introduce noise and lead to incorrect type inference results. Second, type hints may not always exist to reveal the types of variables since many variable usages are not type-revealing. For example, passing a variable to unmodeled external functions cannot provide useful hints to infer the variable's type. Many variable usages only provide indirect type hints, such as the *cmp* instruction, from which we can only infer that two compared variables are of the same type. Additionally, following the existing work [26] to achieve both high precision and scalability, MANTA's underlying points-to analysis could be unsound due to several treatments specified in § 3, potentially leading to some type hints not being captured. Due to the aforementioned reasons, it is possible that correct type hints cannot be captured while incorrect ones are collected, leading to some variables being inferred as incorrect types.

Application Scope. While we have only shown that MANTA's type inference is effective for bug detection, the type analysis could also support other applications. Generally, the inferred type results could benefit many applications that do not strongly rely on soundness. For example, MANTA could assist with binary reverse engineering by increasing decompilation quality or further improve type-based binary fuzzing [39] by providing more precise types to guide seed mutation. However, since the inferred type results cannot guarantee absolute soundness, the system is unsuitable for applications requiring strong soundness guarantees, such as control-flow integrity (CFI) or rigorous program verification.

Generalizability. MANTA's hybrid-sensitive binary type inference approach could be expanded to dynamically typed languages like Python and JavaScript, which also encounter a similar precision-recall trade-off problem. For example, existing type inference methods for dynamically typed languages [6, 61] have observed that many variable types cannot be inferred when there are few static constraints available. To address the problem, they resort to DL-based methods [6, 35, 61] to infer types for more variables. We believe MANTA's hybrid-sensitivity static type inference approach can offer new insights to tackle this problem.

Type Refinement Order. The order of type refinement could affect type inference results. In our early experiments, we observed that flow-sensitive analysis tends to be more aggressive and leads to more type loss. Therefore, we designed our workflow with flow-sensitive refinement placed at the final stage. For instance, if we have an over-approximated type introduced by polymorphic functions, context-sensitive refinement can precisely infer its type, while flow-sensitive refinement may result in the total loss of its type if all the type hints happen to be unreachable on CFG. Similar to the practice of applying hybrid-sensitivity in alias analysis, where heavy-weight analysis is placed in a later stage [31, 41], we chose to place the more aggressive analysis later in MANTA to achieve a better balance between higher precision in type inference and inferring types for more variables.

7 Related Work

Type Inference. Binary type recovery can be reached by static analysis [5, 26, 36, 45, 57], dynamic analysis [49, 69], and machine learning [16, 17, 34, 60, 84]. MANTA is the first tool to leverage rich program structures to refine variable types progressively. As a result, MANTA can overcome the previous limitations of standalone imprecise flow-insensitive type inference [26, 45, 57]. Compared to dynamic tools [49, 50, 69], which recover types from concrete execution traces, MANTA offers several advantages as a static approach, such as independence from concrete inputs and environmental configurations. While some recent works use data-driven approach [16, 17, 34, 60, 84] for type inference, they may suffer from overfitting and predict many incorrect results in unseen binaries. Some works have targeted specific programming languages such as Python [30, 33, 61], JavaScript [7, 40], and JVM [46, 59]. MANTA has a flavor of some of the past efforts [4, 59, 61] based on flow-typing, in which control flows or even calling contexts are respected. Also, MANTA takes a refinement-based approach to refine variable types.

Hybrid Program Analysis. In the field of program analysis, it is a common strategy to organize different analyses in stages, where an imprecise pre-analysis is performed to assist with subsequent precise analyses and improve precision efficiently. Especially for static points-to analysis, the high-level idea of staged analysis has been explored in several forms, including *bootstrapping* [13, 41], *sparse analysis* [32, 73], *client-driven analysis* [31, 70, 71] and *pruning* [28, 48]. Other works on hybrid or selective context-sensitive analysis [42, 51, 58, 74] also share a similar idea. The idea of hybrid analysis has also been applied in dynamic analysis, in which relatively cheap static analysis is performed to reduce the dynamic exploration space in *fuzzing* [37], *testing* [21, 24] or *instrumentation* [14, 56]. However, unlike these approaches, MANTA uses hybrid analysis not for performance but for precision and recall considerations. By aggressively collecting enough type hints through pre-analysis and progressively refining the types, MANTA aims to infer types for more variables while maintaining high precision.

Static Binary Bug Detection. Many static analysis tools are specialized for IoT embedded firmwares [15, 18, 19, 63, 64, 81] or are designed to detect specific types of vulnerabilities [77, 83], leaving only a few tools for general-purpose bug detection [1, 2, 10, 75]. Moreover, none of these tools attempt to infer the types of variables to increase analysis precision, as MANTA does. Some works utilize symbolic execution [20, 22, 68, 82] for bug detection, where Inception [22]

is the most similar to MANTA as it propagates type information from code with debug information to low-level code. However, we do not assume the existence of debug information, setting it apart. Finally, some works detect recurring bugs using binary similarity techniques [23, 27, 80]. In contrast, MANTA is designed to detect zero-day vulnerabilities in binaries and thus does not assume any known bugs.

8 Conclusion

We have described MANTA, our hybrid-sensitive type inference for type-assisted binary bug finding. We have shown that MANTA is precise with high recall, effectively assisting with static bug detection in binary. More excitingly, MANTA has found 86 vendor-confirmed bugs with 64 CVE/PSV IDs assigned.

Acknowledgment

We thank the reviewers for their valuable comments on this work. This work is supported by the PRP/004/21FX grant from the Hong Kong Innovation and Technology Commission and research grants from Huawei and TCL. Yuandao Cai is the corresponding author.

A Artifact Appendix

A.1 Abstract

This section describes the details of the artifact evaluation. The artifact includes a binary analysis tool in the form of binary to perform type inference and bug detection, the testing benchmarks, and the scripts to reproduce the results.

A.2 Artifact check-list (meta-information)

- **Program:** Python; ShellScript; C; C++
- **Binary:** MANTA; RetDec; Z3
- **Data set:** Coreutils; 14 Open-Source Projects
- **Run-time environment:** Ubuntu 20.04
- **Hardware:** x86_64 platform
- **Metrics:** Precision and Recall of type inference and downstream static analysis tasks.
- **Output:** Number similar to Table 3, Table 4, Figure 9, Figure 11, Figure 12, and Table 2 in supplementary materials.
- **Experiments:** Executing prepared several shell scripts and inspecting the output results.
- **How much disk space is required (approximately)?:** 100 G.
- **How much time is needed to prepare workflow (approximately)?:** 1 hour.
- **How much time is needed to complete experiments (approximately)?:** 24 hours.
- **Publicly available?:** No, the tool is used for commercial purpose[5].

[5]More details can be found on https://www.clearblueinnovations.org

A.3 Description

A.3.1 Hardware dependencies. Our experiments were performed on a server with two 20-core Intel(R) Xeon @2.20GHz CPUs and 256 GB DRAM. The evaluators' environment should be comparable to or exceed these specifications to ensure a consistent evaluation.

A.3.2 Data sets. The data sets include coreutils and 14 other open-source projects, all of which are publicly accessible.

A.4 Experiment workflow

There are seven shell scripts inside the main folder to reproduce three major experiments described in the paper. More details can be found in the `README.md` file.

(E0): *[Pre-Process]*

Move to the artifact main folder and execute the preprocessing script:

```
$ ./1_decompile_binary.sh
```

This script will decompile all the binaries into LLVM IRs to be analyzed.

(E1): *[Type Inference] [60 minutes]*

Description: The experiment involves running MANTA on test cases to infer the first-layer type of all the function arguments, corresponding to § 6.1 in the paper. The inferred type results of other type inference tools have already been included in the artifacts. All of the type inference results would be compared with the type on source code to determine precision and recall.

Workflow:

- Execute the script:
  `$ ./2_run_type_inference.sh`
  After roughly 60 minutes, the type inference result of MANTA would be output to corresponding folders.

- Next, execute the script:
  `$ ./3_show_type_inference_result.sh`
  If the previous step succeeds, all the data in Table 3 and Figure 9 will be printed out.

(E2): *[Indirect Call Targets Pruning] [60 minutes]*

Description: The experiment involves utilizing the type inference results produced by different tools to prune away indirect call targets, corresponding to §6.2.1 in the paper. The analysis result would be compared against that generated on LLVM IR of test cases compiled from source code to determine the precision and recall.

Workflow:

- Execute the script:
  `$ ./4_run_indirect_call_pruning.sh`

After roughly 60 minutes, the script would output the result into corresponding folders.

- Execute the script:
  `$ ./5_show_indirect_call_result.sh`
  If the previous step succeeds, all the data corresponding to Table 4 and Figure 11 will be printed out.

(E3): *[Data Dependency Pruning] [16 hours]*

Description: The experiment involves utilizing the type inference results produced by different tools to prune away infeasible data dependency and assist with program slicing for bug detection, corresponding to §6.2.2 in the paper. The analysis result would be compared against that generated on LLVM IR of test cases compiled from source code to determine the FP, FN, precision, and recall.

- Execute the script:
  `$ ./6_run_bug_detection.sh`
  The execution time is estimated to be 16 hours, during which the bug reports on each benchmark assisted with different type inference tools would be generated into corresponding folders. During the period, it is expected that there will be high memory and CPU usage.

- Execute the script:
  `$ ./7_show_bug_detection_result.sh`
  If the previous step succeeds, the script will print out all the data corresponding to Figure 12 inside the paper and Table 2 inside the supplementary material [3].

A.5 Evaluation and expected results

The steps to produce the evaluation results have been discussed in the previous section. Slight variability is to be expected due to different experiment environments, and we recommend running all the experiments in the same environment.

A.6 Experiment customization

Users can edit the script and change the execution path to execute MANTA on other binaries.

A.7 Notes

If the printed-out result misses certain benchmarks, it is expected that a crash occurred during the analysis stage due to OOM, leading to some empty reports. In such cases, users need to delete the corresponding empty file manually and re-execute the corresponding analysis scripts.

References

[1] Cwe-checker. https://github.com/fkie-cat/cwe-checker, 2021. Accessed November 9, 2021.

[2] *QueryX: Symbolic Query on Decompiled Code for Finding Bugs in COTS Binaries (to appear)*, San Francisco, CA, May 2023.

[3] Supplementary material. https://github.com/Ychame/MANTA-Supplement/, 2024.

[4] Michael D. Adams, Andrew W. Keep, Jan Midtgaard, Matthew Might, Arun Chauhan, and R. Kent Dybvig. Flow-sensitive type recovery in linear-log time. *SIGPLAN Not.*, 46(10):483–498, oct 2011.

[5] N. S. Agency. Ghidra reverse engineering tool. https://www.nsa.gov/resources/everyone/ghidra/.

[6] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. Typilus: neural type hints. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 91–105, New York, NY, USA, 2020. Association for Computing Machinery.

[7] Christopher Anderson, Sophia Drossopoulou, and Paola Giannini. Towards type inference for javascript. volume 3586, 07 2005.

[8] D.C. Atkinson. Accurate call graph extraction of programs with function pointers using type signatures. In *11th Asia-Pacific Software Engineering Conference*, pages 326–335, 2004.

[9] Domagoj Babic and Alan J. Hu. Calysto: scalable and precise extended static checking. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 211–220, New York, NY, USA, 2008. Association for Computing Machinery.

[10] Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6), aug 2010.

[11] George Balatsouras and Yannis Smaragdakis. Structure-sensitive points-to analysis for c and c++. In *Sensors Applications Symposium*, 2016.

[12] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010.

[13] Yuandao Cai and Charles Zhang. A cocktail approach to practical call graph construction. *Proc. ACM Program. Lang.*, 7(OOPSLA2), oct 2023.

[14] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, page 39–50, New York, NY, USA, 2008. Association for Computing Machinery.

[15] Libo Chen, Yanhao Wang, Quanpu Cai, Yunfan Zhan, Hong Hu, Jiaqi Linghu, Qinsheng Hou, Chao Zhang, Haixin Duan, and Zhi Xue. Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems. In *USENIX Security Symposium*, 2021.

[16] Ligeng Chen, Zhongling He, and Bing Mao. Cati: Context-assisted type inference from stripped binaries. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 88–98, 2020.

[17] Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Augmenting decompiler output with learned variable names and types. In *USENIX Security Symposium*, 2022.

[18] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. Dtaint: Detecting the taint-style vulnerability in embedded device firmware. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 430–441, 2018.

[19] Kai Cheng, Yaowen Zheng, Tao Liu, Le Guan, Peng Liu, Hong Li, Hongsong Zhu, Kejiang Ye, and Limin Sun. Detecting vulnerabilities in linux-based embedded firmware with sse-based on-demand alias analysis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 360–372, New York, NY, USA, 2023. Association for Computing Machinery.

[20] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: a platform for in-vivo multi-path analysis of software systems. In *ASPLOS XVI*, 2011.

[21] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, page 258–269, New York, NY, USA, 2002. Association for Computing Machinery.

[22] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. Inception: System-wide security testing of real-world embedded systems software. In *USENIX Security Symposium*, 2018.

[23] Yaniv David, Nimrod Partush, and Eran Yahav. Firmup: Precise static detection of common vulnerabilities in firmware. *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[24] David Devecsery, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Optimistic hybrid analysis: Accelerating dynamic analysis through predicated static analysis. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 348–362, New York, NY, USA, 2018. Association for Computing Machinery.

[25] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, page 106–117, New York, NY, USA, 1998. Association for Computing Machinery.

[26] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. Scalable variable and data type detection in a binary rewriter. *SIGPLAN Not.*, 48(6):51–60, jun 2013.

[27] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovre: Efficient cross-architecture identification of bugs in binary code. In *Network and Distributed System Security Symposium*, 2016.

[28] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, page 133–144, New York, NY, USA, 2006. Association for Computing Machinery.

[29] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. *SIGPLAN Not.*, 37(5):1–12, may 2002.

[30] Google. Pytype. https://github.com/google/pytype, accessed 30 May 2023.

[31] Samuel Z. Guyer and Calvin Lin. Client-driven pointer analysis. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, page 214–236, Berlin, Heidelberg, 2003. Springer-Verlag.

[32] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, page 289–298, USA, 2011. IEEE Computer Society.

[33] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. Maxsmt-based type inference for python 3. In *International Conference on Computer Aided Verification*, 2018.

[34] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1667–1680, New York, NY, USA, 2018. Association for Computing Machinery.

[35] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 152–162, New York, NY, USA, 2018. Association for Computing Machinery.

[36] Hex-Rays. The ida pro disassembler and debugger. https://www.hex-rays.com/products/ida/.

[37] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Beacon: Directed grey-box fuzzing with provable path pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 36–50, 2022.

[38] P. Matula J. K˘roustek. Retdec: An open-source machine-code decompiler. Presented at Pass the SALT 2018, Lille, FR, July 2018.

[39] Vivek Jain, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. Tiff: Using input type inference to improve fuzzing. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 505–517, New York, NY, USA, 2018. Association for Computing Machinery.

[40] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In Jens Palsberg and Zhendong Su, editors, *Static Analysis*, pages 238–255, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[41] Vineet Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 249–259, New York, NY, USA, 2008. Association for Computing Machinery.

[42] George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 423–434, New York, NY, USA, 2013. Association for Computing Machinery.

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

[44] Sun Hyoung Kim, Dongrui Zeng, Cong Sun, and Gang Tan. Binpointer: Towards precise, sound, and scalable binary-level pointer analysis. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, CC 2022, page 169–180, New York, NY, USA, 2022. Association for Computing Machinery.

[45] Jonghyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. In *NDSS*, 2011.

[46] Xavier Leroy. Java bytecode verification: Algorithms and formalizations. *Journal of Automated Reasoning*, 30, 08 2003.

[47] Xavier Leroy and Sandrine Blazy. Formal verification of a c-like memory model and its uses for verifying program transformations. *J. Autom. Reason.*, 41(1):1–31, 2008.

[48] Percy Liang and Mayur Naik. Scaling abstraction refinement via pruning. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 590–601, New York, NY, USA, 2011. Association for Computing Machinery.

[49] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. 01 2010.

[50] Ziyi Lin, Jinku Li, Bowen Li, Haoyu Ma, Debin Gao, and Jianfeng Ma. Typesqueezer: When static recovery of function signatures for binary executables meets dynamic analysis. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 2725–2739, New York, NY, USA, 2023. Association for Computing Machinery.

[51] Jiangchao Liu, Jierui Liu, Peng Di, Diyu Wu, Hengjie Zheng, Alex X. Liu, and Jingling Xue. Hybrid inlining: A framework for compositional and context-sensitive static analysis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 114–126, New York, NY, USA, 2023. Association for Computing Machinery.

[52] Zhibo Liu, Yuanyuan Yuan, Shuai Wang, and Yuyan Bao. Sok: Demystifying binary lifters through the lens of downstream applications. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1100–1119, 2022.

[53] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1867–1881, New York, NY, USA, 2019. Association for Computing Machinery.

[54] Alessandro Mantovani, Luca Compagna, Yan Shoshitaishvili, and Davide Balzarotti. The convergence of source code and binary vulnerability discovery – a case study. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '22, page 602–615, New York, NY, USA, 2022. Association for Computing Machinery.

[55] Dr. Paul Muntean, Matthias Fischer, Gang Tan, Zhiqiang Lin, Jens Grossklags, and Claudia Eckert. τcfi: Type-assisted control flow integrity for x86-64 binaries. In *RAID*, 2018.

[56] George C. Necula, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, page 128–139, New York, NY, USA, 2002. Association for Computing Machinery.

[57] Matt Noonan, Alexey Loginov, and David Cok. Polymorphic type inference for machine code. *SIGPLAN Not.*, 51(6):27–41, jun 2016.

[58] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective context-sensitivity guided by impact pre-analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 475–484, New York, NY, USA, 2014. Association for Computing Machinery.

[59] David Pearce and James Noble. Implementing a language with flow-sensitive and structural typing on the jvm. *Electr. Notes Theor. Comput. Sci.*, 279:47–59, 12 2011.

[60] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. Stateformer: Fine-grained type recovery from binaries using generative state modeling. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 690–702, New York, NY, USA, 2021. Association for Computing Machinery.

[61] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. Static inference meets deep learning: A hybrid type inference approach for python. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2019–2030, New York, NY, USA, 2022. Association for Computing Machinery.

[62] Prashant Hari Narayan Rajput, Constantine Doumanidis, and Michail Maniatakos. Icspatch: Automated vulnerability localization and non-intrusive hotpatching in industrial control systems using data dependence graphs. volume abs/2212.04229, 2023.

[63] Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi, Eric Gustafson, Yan Shoshitaishvili, Christopher Krügel, and Giovanni Vigna. Bootstomp: On the security of bootloaders in mobile devices. In *USENIX Security Symposium*, 2017.

[64] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1544–1561, 2020.

[65] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, page 49–61, New York, NY, USA, 1995. Association for Computing Machinery.

[66] Semmle. Codeql. https://securitylab.github.com/tools/codeql, accessed 30 May 2023.

[67] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. *SIGPLAN Not.*, 53(4):693–706, jun 2018.

[68] 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. pages 138–157, 05 2016.

[69] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Network and Distributed System Security Symposium*, 2011.

[70] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, page 387–400, New York, NY, USA, 2006. Association for Computing Machinery.

[71] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, page 59–76, New York, NY, USA, 2005. Association for Computing Machinery.

[72] Yulei Sui and Jingling Xue. Svf: Interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 265–266, New York, NY, USA, 2016. Association for Computing Machinery.

[73] Yulei Sui and Jingling Xue. Value-flow-based demand-driven pointer analysis for c and c++. *IEEE Transactions on Software Engineering*, 46(8):812–835, 2020.

[74] Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. Making pointer analysis more precise by unleashing the power of selective context sensitivity. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[75] Jayakrishna Vadayath, Moritz Eckert, Kyle Zeng, Nicolaas Weideman, Gokulkrishna Praveen Menon, Yanick Fratantonio, Davide Balzarotti, Doup Adam, Tiffany Bao, Ruoyu Wang, Christophe Hauser, and Yan Shoshitaishvili. Arbiter: Bridging the static and dynamic divide in vulnerability discovery on binary programs. In Usenix, editor, *USENIX 2022, 31st USENIX Security Symposium, August 10–12, 2022, Boston, MA, USA*, Boston, 2022. Copyright Usenix. Personal use of this material is permitted. The definitive version of this paper was published in USENIX 2022, 31st USENIX Security Symposium, August 12, 2022, Boston, MA, USA and is available at :.

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

[77] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. 01 2009.

[78] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, page 1–12, New York, NY, USA, 1995. Association for Computing Machinery.

[79] Yichen Xie and Alexander Aiken. Scalable error detection using boolean satisfiability. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 351–363. ACM, 2005.

[80] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. Patch based vulnerability matching for binary programs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 376–387, New York, NY, USA, 2020.

Association for Computing Machinery.

[81] Jiawei Yin, Menghao Li, Wei Wu, Dandan Sun, Jianhua Zhou, Wei Huo, and Jingling Xue. Finding smm privilege-escalation vulnerabilities in uefi firmware with protocol-centric static analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1623–1637, 2022.

[82] Jonas Zaddach, Luca Bruno, Aurélien Francillon, and Davide Balzarotti. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. 02 2014.

[83] Yang Zhang, Xiaoshan Sun, Yi Deng, Liang Cheng, Shuke Zeng, Yu Fu, and Dengguo Feng. Improving accuracy of static integer overflow detection in binary. 11 2015.

[84] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 813–832, 2021.