Protecting Source Code Privacy When Hunting Memory Bugs

Jielun Wu[†], Bing Shui[†], Hongcheng Fan[†], Shengxin Wu[†], Rongxin Wu[‡], Yang Feng[†], Baowen Xu[†], Qingkai Shi[*†]
[†]State Key Laboratory for Novel Software Technology, Nanjing University, China
[‡]School of Informatics, Xiamen University, China
{jielunwu,bingshui,hchfan,wsx}@smail.nju.edu.cn, wurongxin@xmu.edu.cn, {fengyang,bwxu,qingkaishi}@nju.edu.cn

***Abstract*—When proving to a third party that a software system is free from critical memory bugs, software vendors often face the problem of having to reveal their source code, so that the third party can scan the source code using static analysis tools. However, such transparency poses a significant threat to vendors, as the source code typically contains proprietary algorithms, core technical innovations, or trade secrets, exposing them to potential intellectual property risks. In this paper, we present a solution that offers a balance between transparency and code privacy, allowing software vendors to provide minimal source code information while justifying the sufficiency of bug detection. To this end, we propose DIREDUCER, which reduces source code information, a.k.a. debug information, from non-stripped binaries while preserving its utility for memory bug detection. DIREDUCER consists of two components: selective pruning and type minimization. The former eliminates redundant debug information, and the latter is proven to be NP-hard and minimizes type-related debug information by reducing it to the classic set-cover problem, which offers a near-optimal solution. Experimental results show that we can reduce 95% of debug information while maintaining similar bug detection capability compared to using full debug information or the source code.**

I. INTRODUCTION

In situations like the export of critical software, software vendors are typically required to demonstrate that their programs contain no defects, especially severe memory vulnerabilities, such as buffer overruns, to meet legal and regulatory standards. However, proving to the third party that a program is free of such memory bugs is risky of leaking technical or trade secrets because, as reported by BBC [1], software vendors often have to reveal the source code so that the third party can scan the program using static analysis tools [4], [13], [52], [53], [59]. To protect source code privacy, we, as software vendors, may require the third party to utilize bug detectors based on stripped binaries, i.e., binaries without any source code (or debug) information. However, this is often not acceptable because it is well known that detecting program bugs in stripped binaries is not effective due to the lack of critical information, such as data types [54].

In this paper, we aim to provide a solution so that *software vendors can justify providing sufficient source code information for memory bug detection while simultaneously revealing as little source code information as possible.* Our approach, namely DIREDUCER, draws inspiration from data anonymization techniques [17], [32], [44], which are widely used to obscure sensitive information in datasets, making it difficult to trace the anonymized data back to specific individuals or entities. As shown in Figures 1(a) and 1(b), a simple example of data anonymization is the practice of partially omitting sensitive information, such as credit card numbers and billing addresses, from billing data while preserving its verifiability.

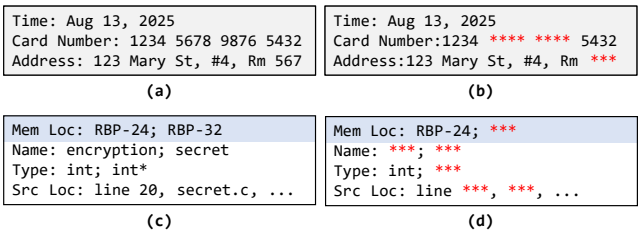


Fig. 1. (a-b) Data anonymization vs. (c-d) DIREDUCER.

DIREDUCER seeks to achieve a similar effect in the domain of memory bug detection over non-stripped binaries. On the one hand, recent research [72] has shown that detecting memory bugs in non-stripped binaries (binaries with debug information) demonstrates similar capability of bug detection at the source-code level. This is essentially because debug information provides source-code information, such as variable types, memory locations, and so on. On the other hand, non-stripped binaries allow for the possibility of protecting source-code privacy by anonymizing as much debug (or source-code) information while preserving the integrity of program logic and the capability of bug detection.

For example, Figure 1(c) demonstrates partial information of variables `encryption` and `secret` contained in the debug information. For memory bug detection, the variable names and their locations in the source files are unnecessary. Only the memory locations and types are needed. Figure 1(d) demonstrates the anonymized variable information, where the variable names and the source files are reduced. Furthermore, `secret` is a variable with type `int`$*$, When the type of `secret` can be inferred (e.g. `int` $*$ `secret` $=$ &`encryption`;), there is no need to preserve its type in the debug information as well. Consequently, we do not need to maintain any debug information for the address `RSP` $-$ `32`, thus achieving source code information protection.

*Qingkai Shi is the corresponding author.

Particularly, DIREDUCER employs two key techniques to anonymize the debug (or the source code) information in non-stripped binaries: selective pruning and type minimization. The former systematically identifies and retains only the portions of debug information that are essential for memory bug detection, thereby eliminating redundant debug information. The latter leverages binary type inference techniques to further remove redundant types. The problem of minimizing type information is formulated as a classic NP-hard problem — the set cover problem, which offers near-optimal solutions for us.

Existing research has partially explored the reduction of debug information and type inference for binary code. Modern compilers can compress debug information to reduce its size [3], [8]. However, such compression does not eliminate any information and, thus, does not offer any protection for source-level privacy. Binary type inference is a critical task in reverse engineering, and numerous studies [24], [29], [35], [40], [48], [58], [66], [67], [71] have focused on inferring understandable type information from stripped binaries, such as strings, pointers, and structures. In contrast, DIREDUCER analyzes the types embedded in the debug information to identify and remove redundant types. Unlike prior work, which focuses on transforming unknown types into explicitly known ones, we transform known types into unknown ones.

Recent studies on privacy-preserving program analysis [30] employed cryptographic techniques, e.g., secure multi-party computation, to preserve privacy. However, these approaches often incur severe performance overhead, making the already inefficient program analyzers (which are well-known to be of high complexity) even worse. Typically, they can be applied to programs with only a few hundred lines of code [30] and, thus, lack practicality. In contrast, DIREDUCER does not introduce apparent overhead to static analyzers.

Obfuscation is another technique that aims to protect source code, but under a completely different philosophy [16], [23], [38], [51]. Obfuscators modify control flows or scramble code, making a program difficult to understand and analyze. For example, an aggressive obfuscator can heavily complicate control flow but reduce the effectiveness of bug detection. Merely obfuscating identifiers (e.g., variable names) offers little benefit, as we demonstrate that completely removing identifier names does not impair bug detection effectiveness. Additionally, obfuscators typically do not operate on debug information, whereas DIREDUCER attempts to minimize debug information as much as possible. In other words, obfuscation is orthogonal to our approach as we can still apply obfuscation to enhance privacy after applying DIREDUCER.

In summary, we make the following contributions:

- We provide a solution where software vendors can justify providing sufficient source code information for memory bug detection while simultaneously revealing as little source code information as possible.
  - We propose a novel debug information reduction technique, incorporating selective pruning and type minimization to prevent source code privacy leakage during static memory bug detection.
  - We reduce the type minimization problem to the classic set cover problem, which establishes the problem's difficulty, i.e., NP-hardness, but provides near-optimal solutions to prune debug information.
- We evaluate DIREDUCER on a broad set of programs, showing that DIREDUCER can reduce 94.55% debug information on average, revealing sufficient but minimal source code information for bug detection. The artifact is publicly available [5].

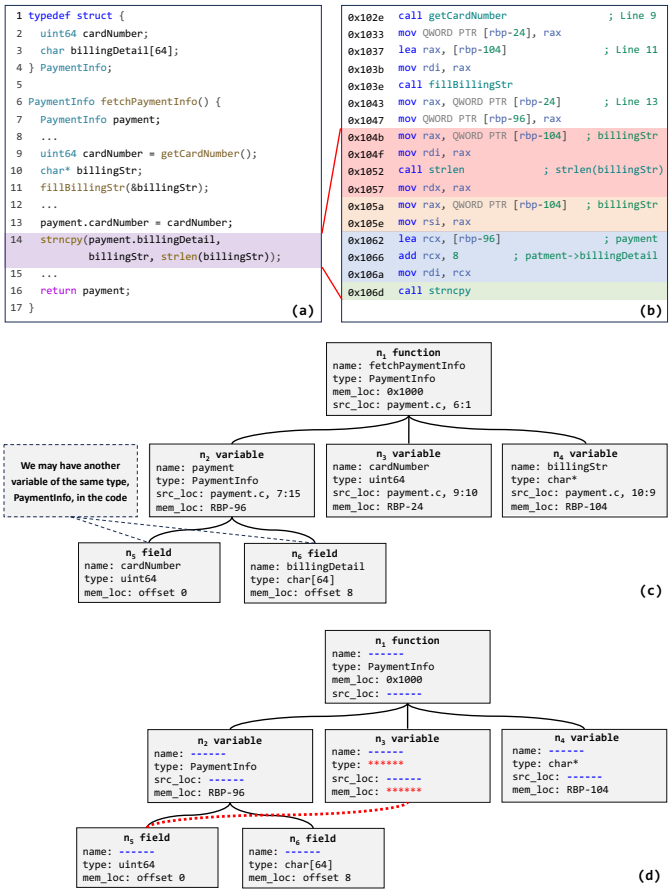


Fig. 2. (a) Example code of a payment transaction system. (b) The x86 assembly code. (c) Original DIG. (d) Reduced DIG.

II. Background and Overview

A. Debug Information Demystified

Non-stripped binaries contain debug information in various formats, such as PDB [11] and DWARF [6]. PDB is predominantly used on Windows platforms, whereas DWARF is the de facto standard across most other systems. Debug information, in either PDB or DWARF format, is a graph structure.

Definition 1 (Debug Information Graph (DIG)). *The debug information in a binary is a graph $G = (N, E)$, where*

- *Each node in N represents an entity, such as functions and variables, and consists of attribute-value pairs;*
- *Each edge in $E \subseteq N \times N$ represents hierarchical relationships between the entities.*

TABLE I: Classification of the Debug Information.

Classification	Description	DIG Attribute
1. Symbol Name	The names of program entities, such as function names and variable names.	`name`
2. Building Config	Compilation options, building directories, linking parameters, and so on.	`config`
3. Source Location	Specify the source location of program entities, including the source file, source line, and source row.	`src_loc`
4. Memory Location	Specify the address of program entities, in memory or registers.	`mem_loc`
5. Type Metadata	primitive types (e.g., int, char) and compound types (e.g., structure), and the memory layout of types.	`type`

TABLE II: Memory Bugs and Related Types.

Bug Category	Related Types
Null Pointer Dereference [14], [61]	Numeric, Pointer, Structure, ...
Memory Leak [42]	Numeric, Pointer, Structure, ...
Use-After-Free [33]	Numeric, Pointer, Structure, ...
Double-Free [21], [65]	Numeric, Pointer, Structure, ...
Buffer Overflow [46], [70]	Array, Numeric, Pointer, ...
VTable Escape [27], [69]	Class, VTable Pointer, ...
Type Confusion [34], [39], [68]	Class, Pointer, Union, ...

Example 1. Figure 2(a) presents a simplified code snippet of a compilation unit `payment.c` from a payment system. It includes a function `fetchPaymentInfo`, a structure type `PaymentInfo`, two other types (i.e., `uint64` and `char`[64]), and three variables: `payment`, `cardNumber`, and `billingStr`.

Figure 2(c) shows the DIG embedded in the corresponding binary code in (b). The DIG consists of six nodes, n_1 through n_6. The node n_1 represents the function `fetchPaymentInfo` and is connected to three variable nodes, n_2, n_3, and n_4, as these variables are defined within the scope of the function. Particularly, the node n_2 represents a structure-type variable with two fields denoted by n_5 and n_6. All variables of the same compound type (e.g., structure) share the field nodes, as illustrated in the figure. Each node may have many attributes, including `name`, `config`, `src_loc`, `mem_loc`, and `type`, representing the symbol name, building config, source location, memory location, and type metadata, as outlined in Table I.

Exposing the source code information in DIG may pose serious security threats. For example, leaking building configuration information can reveal compilers and third-party libraries used in the program, enabling attackers to exploit known issues in them [15], [19], [28]. Disclosing symbol names or type metadata may expose the intent of the program [22] or even trade secrets. Revealing the source location could expose the code organization structure, which is private for software vendors. Additionally, leaking memory location may allow attackers to launch targeted attacks, such as virtual function table hijacking [69].

To measure the amount of source code information exposed by a binary file, we define the following source exposure metric quantified by simple attribute counting.

Definition 2 (Source Exposure Metric). *Given an original DIG G and a reduced DIG G', the source exposure metric $\mathcal{P}(G', G) = |Attr'|/|Attr| \times 100\%$, where $|Attr|$ and $|Attr'|$ are the number of all attributes in G and G', respectively.*

Example 2. Figure 2(d) is a reduced DIG, G', where attributes masked by '-' and '*' are removed. Compared to the original DIG G in (c), the reduced DIG exposes $\mathcal{P}(G', G) = 10/22 = 45.5\%$ of the source code information.

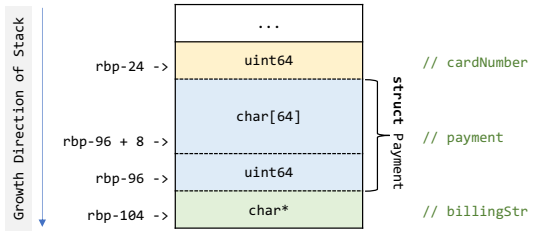


Fig. 3. Stack Layout from Debug Information.

B. Memory Bug Detection

Memory bugs are among the most common and severe software security vulnerabilities in the software world [18]. Table II lists the common memory bugs. We observe that detecting almost all these memory bugs requires only knowing the data type of memory regions.

Example 3. Identifying memory bugs like buffer overflow necessitates the knowledge of memory layout and array types. The code in Figure 2(a) is potentially vulnerable to buffer overflow, with Figure 2(b) presenting its assembly code. If `billingStr` initialized at line 11 exceeds 64 bytes, the invocation of `strncpy` at line 14 will cause a buffer overflow because `payment.billingDetail` is of the type `char`[64]. With the array types and memory locations in DIG, we can get the stack layout as Figure 3, which illustrates how local variables of different types, including the vulnerable array `payment.billingDetail`, are located on the stack. Without types and memory locations in the DIG, the stack layout is unknown. Then, we cannot locate the array or determine its size and, thus, cannot detect the buffer overflow. In contrast, debug information, such as the variable names and source code location, is not important for memory bug detection.

The observation above (i.e., bug detection could relate only to a little source code information like types) provides an opportunity to significantly reduce source code information from non-stripped binaries while still enabling effective memory bug detection. As such, we can achieve the following goal.

> **Goal**: When software vendors are required to demonstrate that their programs are free of memory bugs, they can then reveal as little source code information as possible while simultaneously justifying that they have provided sufficient source code information for memory bug detection.

C. Approach in a Nutshell

As shown in Figure 4, DIREDUCER reduces source code information in four steps: **(1)** Taking a non-stripped binary as input, constructing a control flow graph (CFG), and separating

the debug information from the binary. **(2)** Building DIG and employing selective pruning to remove irrelevant source code information, yielding a pruned DIG. **(3)** Constructing a type dependency graph (TDG) based on the CFG and the pruned DIG from the previous step, and then performing type minimization based on the TDG to further reduce debug information. **(4)** Reconstructing the reduced debug information with the stripped binary to produce the final binary as the output. The practical implementation of the above workflow must overcome several challenges, which, together with our solutions, are briefly discussed below.

Selective Pruning. This step aims to identify the source code information unrelated to memory bug detection from more than 400 kinds of debug information in DIG [6], [7]. As mentioned earlier, memory bug detection relies on just a little source code information, such as the memory locations and types of program variables. Thus, the challenge below arises.

> **Challenge 1**: What key elements within numerous kinds of debug information are valuable for memory bug detection?

Given that not all debug information contributes to the effectiveness of memory bug detection, we systematically identified the debug information that needs to be preserved after reviewing the DWARF documentation and the implementations of relevant compilers. Consider the DIG in Figure 2(c). This step prunes `name` and `src_loc` in all nodes (n_1 through n_6), which, as discussed in Example 3, are not related to the bug in the code, yielding a source exposure metric $\mathcal{P}(G', G) = 54.5\%$.

Type Minimization. The following challenge asks whether we should retain all types, even though types are critical source code information for memory bug detection. We observe that retaining only a subset of types in debug information is sufficient for memory bug detection.

> **Challenge 2**: How can we minimize types while preserving the effectiveness of memory bug detection?

To address this challenge, we propose a type minimization technique to identify and remove redundant types embedded in DIG. Intuitively, assuming that the set of types embedded in DIG is Θ, and the set of types after reduction is Θ', we ensure that all types in the set Θ can be inferred from set Θ' in some manner, such that we can justify that the retained debug information is sufficient for detecting memory bugs. To this end, we reduce the type minimization problem to a classic set cover problem, which, although NP-hard, provides an efficient approximation algorithm for a nearly optimal solution.

Consider the DIG in Figure 2(c). We can remove the type `uint64` in the node n_3 by our type minimization technique, because these types can be inferred from the types embedded in n_5. To facilitate type minimization, we extend DIG as Figure 2(d) by adding red dashed lines (between n_5 and n_3) to indicate that one type can be inferred from others.

After type minimization, since the node n_3 no longer contains any information (e.g. `type`, `mem_loc`), it can be entirely removed. Consequently, the source exposure metric $\mathcal{P}(G', G)$ is further reduced from 54.5% to 45.5%. In practice, according to our evaluation, we can reduce the metric to 5.45% on average, meaning that only a little source code information has to be revealed.

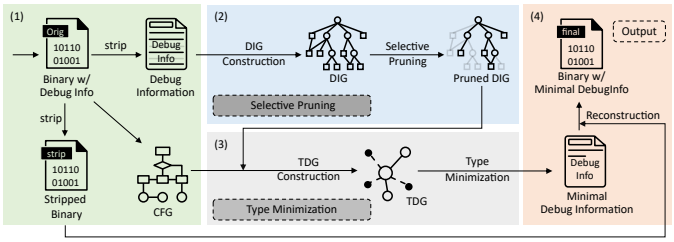


Fig. 4. Overview of DIREDUCER.

III. Selective Pruning

This step aims to prune all debug information irrelevant to memory bug detection, such as the DIG attributes `name`, `config`, `src_loc`, `mem_loc`, and `type`, listed in Table I. While this step appears straightforward, it is particularly challenging because, in practice, an abstract attribute, e.g., `type`, corresponds to a number of concrete attributes in the DWARF standard, and we must identify all useful ones from among thousands of concrete attributes. To determine whether or not to retain particular debug information, we conducted extensive empirical investigations, including reviewing DWARF documentation and implementations in relevant compilers (e.g., gcc, clang) to check if it is related to memory bug detection. As a result, for the first time to the best of our knowledge, we systematically cataloged the DWARF attributes valuable for memory bug detection in Table III, which includes two kinds of debug information indicating the memory locations storing program variables and the variable types.

Since DWARF is the de facto standard for debug information, the manual efforts in this step only need to be carried out once. Thus, we do not try to automate this step.

Memory Locations. The first group of attributes in Table III specifies where program variables or functions are stored in memory, including seven attributes in total. The attribute `DW_AT_low_pc` records the entry point of a function, while the attribute `DW_AT_high_pc` specifies its length. Such information is particularly valuable for fundamental binary analysis techniques like disassembly, as disassemblers cannot always accurately identify function boundaries [47].

The attribute `DW_AT_frame_base` records the stack pointer's position, which is essential for computing variable memory location since locations are stack-pointer-relative. The `DW_AT_location` attribute specifies the memory location of a variable, while `DW_AT_return_addr` records the memory location storing the function return address, enabling static analyzers to detect potential overflow bugs that may maliciously rewrite the return address. When a variable is on the stack, its location is computed based on both `DW_AT_frame_base` and `DW_AT_location`. When a variable is in register, its location can be computed based on `DW_AT_location` only.

The attribute DW_AT_data_member_location specifies the offset of a field relative to its parent compound type's base address. The attribute DW_AT_vtable_elem_location specifies the offset of a virtual function table pointer relative to its class's base address in class types.

Example 4. In Figure 2(c), the attribute specifying memory location, mem_loc, in each variable node is computed based on DW_AT_frame_base and DW_AT_location. For example, the variables payment, cardNumber, and billingStr are stored at memory location RBP − 96, RBP − 24, and RBP − 104 respectively, where RBP is computed as per DW_AT_frame_base and the offsets to RBP are computed from DW_AT_location.

Types. While the debug information of memory locations specifies a bounded memory region for each program variable, the type metadata in Table III indicates the layouts and data types of the memory region. Attributes prefixed with DW_TAG denote specific types. For instance, DW_TAG_base_type represents primitive types such as int, long, float, etc. The others like DW_TAG_structure_type specify composite types derived from the primitive types. In addition, attributes prefixed with DW_AT provide detailed type specifications. For example, the attribute DW_AT_byte_size indicates the size of a type in bytes, and the attribute DW_AT_upper_bound specifies the number of elements in an array.

Example 5. In Figure 2(c), querying the attribute DW_AT_type of the variable payment reveals its type as PaymentInfo. Further inspection of the attributes, DW_TAG_structure_type, confirms that payment is a structure-type variable containing two fields: cardNumber and billingDetail. Moreover, the field billingDetail is annotated with the attribute DW_AT_data_member_location, which records the field's offset of 8 bytes relative to the structure's base address.

As discussed before, the selective pruning step removes all attributes not in Table III from the DIG nodes. In Figure 2(d), attributes masked by '-' indicate they were removed during the selective pruning process. As shown, all attributes such as name and src_loc have been eliminated, as these attributes are unrelated to memory bug detection. In contrast, attributes related to memory bug detection, such as type and mem_loc, are preserved. These attributes correspond to those enumerated in Table III and will be further anonymized in the next step.

To conclude, this step makes two contributions. First, to our knowledge, this represents the first systematic study that (1) analyzes mainstream debug information standards and (2) establishes a taxonomy of debug information. Second, it is a necessary step in our overall solution, which enables software vendors to provide sufficient source code information for memory bug detection while simultaneously revealing as little source code information as possible.

TABLE III: DWARF Debug Information for Memory Bug Detection

Classification	Debug Info Attribute	Description
Memory Location	DW_AT_low_pc	Function entry point.
	DW_AT_high_pc	Function length.
	DW_AT_frame_base	Function stack base.
	DW_AT_location	Variable memory address.
	DW_AT_return_addr	Subroutine return address.
	DW_AT_data_member_location	Data member offset.
	DW_AT_vtable_elem_location	Virtual function vtable slot.
Type Metadata	DW_TAG_base_type	Indicates a base type.
	DW_TAG_union_type	Indicates a union type.
	DW_TAG_structure_type	Indicates a structure type.
	DW_TAG_enumeration_type	Indicates an enumeration type.
	DW_TAG_subroutine_type	Indicates a subroutine type.
	DW_TAG_array_type	Indicates an array type.
	DW_TAG_class_type	Indicates a class type.
	DW_TAG_reference_type	Indicates a reference type.
	DW_TAG_pointer_type	Indicates a pointer type.
	DW_AT_byte_size	Size of the type in bytes.
	DW_AT_type	Reference of other types.
	DW_AT_bit_size	Size of the type in bits.
	DW_AT_upper_bound	Upper bound of the array.
	DW_AT_encoding	Encoding of primitive types.

IV. Type Minimization

From the perspective of software vendors, we should reduce as much debug (or source code) information as possible, even though some information may be theoretically recoverable. To this end, type minimization allows vendors to reduce more debug information, while still justifying that the revealed source code information is sufficient for memory detection. The basic idea is that all types and memory locations in debug information that can be theoretically recovered from retained debug information can be removed. This section elaborates on this idea in two steps: the data structure (§IV-A) and the algorithm (§IV-B) for type minimization.

A. Type Dependency Graph

Abstract Program. To ease the explanation, we use the abstract program in Figure 5 to describe common binary programs (e.g., x86 binaries). At the top level, a program is viewed as a sequence of instructions. We abstract x86 assembly instructions into five categories: copy, load, store, binary, and merge instructions. Each instruction has several operands, which are an arithmetic expression over a register, denoted as o, or a memory location whose address is o, denoted as $[o]$. These abstract instructions have counterparts in real x86 assembly. For example, mov eax, ebx corresponds to a copy instruction eax = ebx. In addition, we also introduce an extra merge instruction, which merges multiple definitions of the same variable from different paths. Each operand in a binary instruction must be a primitive type, such as a number or a pointer. From the perspective of source code or debug information, these primitive types can be combined into a compound type as shown in Figure 6.

Type Dependency Graph. For type minimization, i.e., reducing the types that can be inferred from others, we define the type dependency graph as below.

Definition 3 (**Type Dependency Graph (TDG)**). *Given a DIG $G = (N, E)$, a type dependency graph is an undirected graph $G' = (N', E')$ where*

- *For each DIG node $n \in N$ of a primitive type, if it denotes a variable, $n \in N'$; if it denotes a field of a compound-type variable, we have a TDG node $n' \in N'$ which is a copy of n for that variable. Each node $n' \in N'$*

$$
\begin{array}{rcll}
\textit{Program } \mathbf{P} & ::= & \mathbf{I}+; & \\
\textit{Instruction } \mathbf{I} & ::= & o_1 = o_2 & \textbf{::copy} \\
 & & \mid [o_1] = o_2 & \textbf{::store} \\
 & & \mid o_1 = [o_2] & \textbf{::load} \\
 & & \mid o_1 = o_2 \lozenge o_3 & \textbf{::binary} \\
 & & \mid o_1 = \Phi(o_2, o_3, \ldots) & \textbf{::merge} \\
\textit{Operand } o & ::= & r \mid r + num \mid r - num & \\
 & & & \\
\lozenge & \in & \{+, -, \times, \div, <<, \ldots\} & \\
r & \in & \textbf{Registers} & \\
num & \in & \mathbb{Z} &
\end{array}
$$

Fig. 5. Abstract program in DIREDUCER.

$$
\begin{array}{rcl}
\textit{Type } \mathbf{T} & ::= & \mathbf{T}_{prim} \mid \mathbf{T}_{comp} \\
\textit{Primitive Type } \mathbf{T}_{prim} & ::= & \text{int} \mid \text{uint} \mid \text{float} \mid \ldots \mid \mathbf{T}* \\
\textit{Compound Type } \mathbf{T}_{comp} & ::= & \{\mathbf{T}_1 : \text{offset}_1, \mathbf{T}_2 : \text{offset}_2, \ldots\}
\end{array}
$$

Fig. 6. Typing in DIREDUCER.

essentially consists of a pair of a memory location and the type of the memory location, because other debug information has been removed from DIG nodes.

- *Each undirected edge in $E' \subseteq N' \times N'$ represents a type dependency, which will be discussed later.*

Example 6. In Figure 2(d), the DIG nodes n_3 and n_4 denote primitive-type variables. Thus, they belong to the TDG. Furthermore, DIG nodes n_5 and n_6 represent primitive-type structure fields and can be referenced by multiple variables. Thus, we create copies of n_5 and n_6 in the TDG for each variable referring to the fields. The copies in TDG aim to distinguish the structure fields of different variables.

TDG Construction. Algorithm 1 shows how we build a TDG, $G' = (N', E')$. Lines 2–3 initialize the nodes and edges. Lines 5–9 build the TDG edges, i.e., the dependency relation between two nodes. Given an instruction, Line 5 determines if the two operands used in the instruction depend on each other in terms of types, according to Table IV. If there exists a dependency relation, Lines 6-7 acquire the nodes of the operands by calling the procedure `Node`$(\cdot)$: if there is a corresponding TDG node, the node is returned; otherwise, a new fake node is created and returned. Since each operand must be of a primitive type in a binary instruction, if the operand has a corresponding DIG node in the debug information, the corresponding TDG node must have been added to the node set in Line 2. Thus, we have the code comments between lines 5 and 6. Line 8 then adds the edge to the TDG. After all instructions are processed, line 9 iterates and removes every fake node by connecting its predecessors and successors.

Table IV features a simple type dependency analysis, which defines dependency relations for five types of instructions. For `copy`, `load`, `store`, and `merge` instructions, it is straightforward to determine that there is a dependency relation between the operands. For `binary` instructions, it is necessary to determine whether the operands are pointers. For example, consider the instruction `eax` $=$ `ebx` $+$ `edx`. If `ebx` and `edx` hold a pointer and an integer, respectively, we can infer that `eax` also holds a pointer based on `ebx`. Therefore, we only establish dependency between `eax` and `ebx`.

Algorithm 1: TDG Construction

```
1  procedure BuildTDG(P)
2      N' ← all nodes created based on Definition 3;
3      E' ← ∅;
4      for instruction I ∈ P do
5          if operand o1 relies on o2 in I as per Table IV then
               /* n1, n2 ∈ N' or are fake nodes */
6              n1 ← Node (o1, N');
7              n2 ← Node (o2, N');
8              E' ← E' ∪ {(n1, n2)};
9      remove each fake node by connecting its predecessors and
         successors;
10     return (N', E');
11 procedure Node(o, N')
12     if there exists a corresponding node n for o in N' then
13         return n;
14     return fake node for o;
```

TABLE IV: Connection Rules of Instruction

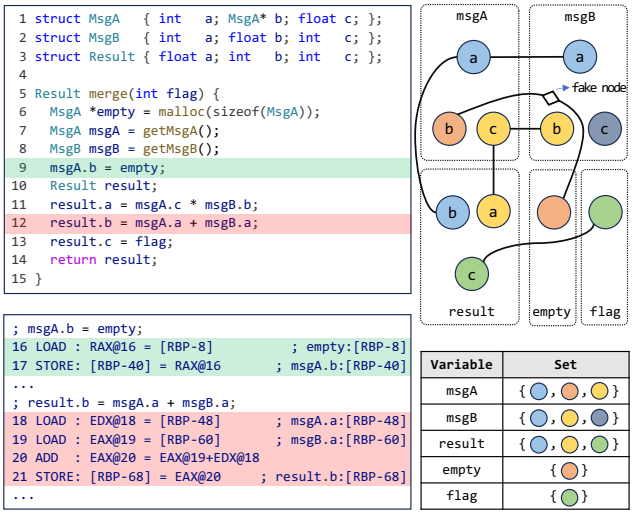
Instructions	Connection Rules
copy: $o_1 = o_2$	rely_on(o_1, o_2)
store: $[o_1] = o_2$	rely_on(o_1, o_2)
load: $o_1 = [o_2]$	rely_on(o_1, o_2)
binary: $o_1 = o_2 \lozenge o_3$	① $\neg$`IsPtr`$(o_1) \Rightarrow \forall i > 1$: rely_on(o_1, o_i)
	② `IsPtr`$(o_1) \wedge$ `IsPtr`$(o_2) \Rightarrow$ rely_on(o_1, o_2)
	③ `IsPtr`$(o_1) \wedge$ `IsPtr`$(o_3) \Rightarrow$ rely_on(o_1, o_3)
merge: $o_1 = \Phi(o_2, o_3, \ldots)$	$\forall i > 1$: rely_on(o_1, o_i);

Example 7. Figure 7(a) shows a source code snippet and its corresponding assembly code using the syntax in Figure 5. Figure 7(b) shows the TDG, in which each circle node is a DIG node denoting a primitive-type variable or a field of a compound type. Multiple circle nodes may belong to the same variable, e.g., `msgA`, and, thus, are placed in the same dashed box. The rhombus node in the TDG is a fake node created during TDG construction, which is removed at the end of the TDG construction algorithm.

Consider the source code `msgA.b` $=$ `empty` and its corresponding assembly in green. In the assembly code, we use a suffix after a register, e.g., `RAX@16`, to denote the register defined at line 16. When using Algorithm 1 to process the assembly code and visit line 16, we create a fake node for `RAX@16` because it does not correspond to any DIG node. We also make an edge between the fake node and the DIG node of [`RBP` $-$ `8`], which represents the variable `empty` and, thus, is placed in the dashed box labeled `empty` in Figure 7(b).

Similarly, when processing the assembly instruction at line 17, we connect the fake node representing `RAX@16` and the node of [`RBP` $-$ `40`], which represents the field `msgA.b`. At the end, Line 9 in Algorithm 1 removes all fake nodes.

Complexity. The procedure for TDG construction in Algorithm 1 is of linear time complexity with respect to the program size. Essentially, the algorithm consists of two steps. The first step, lines 4-8, processes each instruction once in constant time. Thus, this step has linear complexity. The second step is in line 9, which removes fake nodes from the graph. Since the number of fake nodes is linear in the number of instructions and removing each fake node takes constant



Variable	Set
msgA	{, , }
msgB	{, , }
result	{, , }
empty	{ }
flag	{ }

Connected component	Variable fields	Connected component	Variable fields
cc_1	{msgA.a, msgB.a, result.b}	cc_4	{result.c, flag}
cc_2	{msgA.b, empty}	cc_5	{msgB.c}
cc_3	{msgA.c, msgB.b, result.a}	/	/

(c)

Fig. 7. (a) C language example code and corresponding abstract program instructions for TDG Construction. (b) The TDG corresponding to the example code. (c) Connected components in the TDG.

time, this step also has linear complexity. Thus, in total, the complexity of Algorithm 1 is linear in the program size.

B. Type Minimization

Recall that type minimization aims to reduce redundant (theoretically recoverable) `type` and `mem_loc` from debug information. However, in practice, we cannot remove redundant `type` and `mem_loc` in DIG nodes denoting fields within a compound type, e.g., n_5 in Figure 2, as these nodes may be referenced by other variables. Their removal would yield DIGs non-compliant with the DWARF standard. Instead, we can only remove `type` and `mem_loc` in a DIG node denoting a variable, e.g., n_3 in Figure 2. As such, the objective of type minimization becomes to find the minimum number of variable nodes such that their `type` and `mem_loc` attributes can infer all other variables' `type` and `mem_loc`, which then can be removed from DIG. If the nodes removed are of primitive types, we can fully recover their `type` and `mem_loc`. If the nodes removed are of compound types, we can recover the types at the granularity of primitive types, i.e., recovering the `type` and `mem_loc` for all fields within a compound type. In what follows, we formally define the problem of type minimization, show that this is an NP-hard problem, and provide a suboptimal solution.

Formalization. Recall that each TDG node is a DIG node (or its copy) of a primitive type, and each TDG edge represents a type dependency. Types at both ends of an edge can be mutually inferred. As such, we can divide a TDG into a set of connected components (CCs), denoted as $C = \{cc_1, cc_2, \dots\}$, where the type of any node within a CC can infer the types of all other nodes in that same CC. For example, the TDG in Figure 7(b) contains five CCs, denoted as cc_1, cc_2, cc_3, cc_4, and cc_5. Elements in each CC are illustrated in Figure 7(c).

As illustrated in Figure 7, a program variable, say $v \in V$, may cover multiple connected components, denoted as $\nabla(v) = \{cc_1, cc_2, \dots\}$. For example, as shown in Figure 7, $\nabla(\texttt{msgA}) = \{cc_1, cc_2, cc_3\}$ because cc_1, cc_2, and cc_3 contain `msgA.a`, `msgA.b`, and `msgA.c`, respectively. Based on the above data structure, the type minimization problem is defined below.

Definition 4 (Type Minimization Problem). *Let $C = \{cc_1, cc_2, \dots\}$ be the set of CCs and $V = \{v_1, v_2, \dots\}$ be the set of variables. Our goal is to determine the minimal variable set $V_{min} \in V$ such that*

$$\bigcup_{v_i \in V_{min}} \nabla(v_i) = C$$

where $\nabla(v_i)$ denotes the set of CCs connected to v_i.

Example 8. *In Figure 7, V = {`msgA`, `msgB`, `result`, `empty`, `flag`}, and V_{min} can be {`msgB`, `empty`, `flag`}. V_{min} allows us to infer the types of each field of `msgA` and `result` independently along the edges in the TDG. Since we recover the field types independently, compound types cannot be reassembled into their original composite form, thereby providing a degree of protection for source code privacy.*

NP-Hardness. To prove the NP-hardness of the type minimization problem, we construct a polynomial-time reduction from the classic set cover problem, which is NP-hard, to type minimization. The set cover problem is defined below.

Definition 5 (Set Cover Problem). *Let $U = \{u_1, u_2, \dots\}$ be a universal set and $\mathbb{S} = \{S_1, S_2, \dots\}$ such that each $S_i \subseteq U$ and $\bigcup_{S_i \in \mathbb{S}} S_i = U$. The goal is to find $\mathbb{S}_{min} \subseteq \mathbb{S}$ satisfying*

$$\bigcup_{S_i \in \mathbb{S}_{min}} S_i = U$$

where $|\mathbb{S}_{min}|$ is minimized.

Example 9. Given the universe set $U = \{u_1, u_2, u_3, u_4, u_5\}$ and $\mathbb{S} = \{\{u_1, u_2\}, \{u_3, u_4, u_5\}, \{u_2, u_3\}\}$, we need to select at least two subsets, such as $\mathbb{S}_{min} = \{\{u_1, u_2\}, \{u_3, u_4, u_5\}\}$, to cover all elements in U.

Theorem IV.1. *The Type Minimization Problem is NP-hard.*

Proof. We prove the existence of the reduction by constructing a mapping from the set cover problem to the type minimization problem. Specifically, We map the universal set $U = \{u_1, u_2, \dots\}$ in the set cover problem to connected components set $C = \{cc_1, cc_2, \dots\}$ in the type minimization problem. For each variable v_i, its associated set of connected components $\nabla(v_i)$, corresponds to element S_i in the collection $\mathbb{S}$ of the set cover problem. Let $\nabla = \{\nabla(v_1), \nabla(v_2), \dots\}$, then collection $\mathbb{S}$ can be directly mapped to ∇. Through the above mapping from the set cover problem to the type minimization problem, we observe that both share the same optimization

Algorithm 2: Type Minimization

```
1 procedure minimize(C, V)
2     V_min ← ∅; C' ← C ;
3     while C' ≠ ∅ do
4         Select the set ∇(v_i), v_i ∈ V \ V_min that covers the most
            uncovered CCs of C';
5         V_min ← V_min ∪ {v_i} ;
6         C' ← C' \ ∇(v_i) ;
7     return V_min ;
```

objective. In other words, the type minimization problem is at least as hard as the set cover problem, thereby establishing its NP-hardness. □

An $H(|C|)$-Approximation Solution. The discussion above reduces the set cover problem to type minimization, demonstrating its NP-hardness. In what follows, we discuss the reduction from type minimization to the set cover problem, demonstrating their computational equivalence and allowing us to reuse existing suboptimal solutions.

Theorem IV.2. *The type minimization problem is computationally equivalent to the set cover problem and has an $H(|C|)$-approximation solution.*

Proof. Given the structural similarity between the type minimization problem and the set cover problem, this reduction follows an analogous procedure. We map the set of connected components C in the type minimization problem to the universal set U in the set cover problem. Similarly, the set $\nabla = \{\nabla(v_1), \nabla(v_2), \ldots\}$ is mapped to the collection $\mathbb{S}$ in the set cover problem. The above mapping establishes a polynomial-time reduction from the type minimization problem to the set cover problem. □

Given the theorem above, we can apply existing sub-optimal approximation algorithms for the set cover problem to address our type minimization problem. Particularly, we use the classic greedy algorithm [37]. Algorithm 2 outlines the procedure for greedily selecting the variables to cover all the CCs. In the algorithm, we create two sets V_{min} and C', where V_{min} tracks the variable we choose and C' represents the CCs remaining to be covered by the variables. Line 2 initializes the minimal variable set V_{min} to empty and sets C' to C, meaning that all CCs have not been covered. Each iteration in the loop (Lines 4-6) chooses one variable v_i to cover the most CCs in C' until all CCs are covered.

The greedy algorithm is an $H(|C|)$-approximation algorithm, where $|C|$ is the number of connected components in TDG, and $H(n)$ is the n-th harmonic number, i.e., $H(n) = \Sigma_{k=1}^{n} 1/k \leq \ln n + 1$. In other words, the greedy algorithm will always find a cover at most $H(n)$ times as large as the optimal cover [37]. The time complexity of the greedy algorithm is almost linear [37], i.e., $O(|C| \log |C|)$. Such an almost linear complexity ensures its efficiency in practice. In the next section, we provide a comprehensive evaluation of its effectiveness and efficiency.

TABLE V: Results of Debug Information Reduction.

Prgram	KLoC	Before		After (DIREDUCER)		$P(G', G)$
		Size (MB)	$\|Attr\|$ ($\times 10^5$)	Size (MB)	$\|Attr\|$ ($\times 10^5$)	
perlbench	710	2.60	5.99	0.24	0.60	11.53%
gcc	2,576	16.9	47.9	0.96	2.32	4.84%
mcf	7	0.04	0.09	0.00	0.01	7.67%
omnetpp	647	9.70	24.1	0.47	1.12	4.63%
xalancbmk	1,632	33.5	80.4	1.30	3.27	4.47%
x264	145	0.82	2.17	0.11	0.32	14.82%
deepsjeng	27	0.17	0.40	0.01	0.02	5.14%
leela	92	1.80	4.48	0.08	0.19	4.24%
xz	46	0.30	0.69	0.04	0.10	13.98%
sqlite3	553	1.70	2.78	0.22	0.53	19.20%
openssl	90	0.53	1.19	0.04	0.10	8.59%
nginx	204	2.20	7.86	0.25	0.75	9.53%
git	923	6.40	18.7	0.56	1.38	7.39%
libiconv	34	0.17	0.27	0.02	0.05	19.51%
curl	40	0.40	1.17	0.03	0.07	6.50%
Avg	515	5.14	13.21	0.29	0.72	**5.45%**

V. EVALUATION

We implemented DIREDUCER on top of Capstone [2] and LibDWARF [9] for x86-64 binaries. The former is a disassembly framework that allows us to analyze binary instructions. The latter is a standard library for manipulating debug information. With the tool in hand, we conduct experiments to address the following three research questions regarding the goal of this work: revealing as little debug (or source code) information as possible while still being able to justify that the revealed information is sufficient for memory bug detection.

- **RQ1 (Effectiveness).** How effective is DIREDUCER in reducing source code exposure?
- **RQ2 (Efficiency).** How efficient is DIREDUCER in reducing source code exposure?
- **RQ3 (Sufficiency).** Does DIREDUCER retain sufficient debug information for memory bug detection?

Benchmarks. As shown in Table V, our experimental benchmarks include two categories of C/C++ programs: SPEC CPU 2017 INT [10], which is widely used in the research community, and a set of commonly-used real-world programs. The sizes of these programs are quite representative, ranging from 7 KLoC to 2,576 KLoC, with 515 KLoC on average. All programs are compiled into binaries using GCC 13.2.0.

Evaluation Method. Since the problem we try to address is quite challenging, to the best of our knowledge, there is only one work closely related to ours, as it also tries to protect source code privacy during program analysis but based on zero-knowledge proofs [30]. However, this technique is too expensive and can only analyze small code snippets with only a few hundred lines of code, thus failing to analyze all our benchmark programs. Thus, we cannot empirically compare it to DIREDUCER. Instead, this section aims to demonstrate the extent to which our goal is achieved through the three RQs.

Environment. All experiments were conducted on a server running Ubuntu 22.04 with 256 GB of memory and two Intel Xeon Gold 6430 processors for 64 physical cores.

A. RQ1 - Effectiveness

Effectiveness. Table V lists the sizes (in MB) of debug information and the number of attributes (in $\times 10^5$) in DIG before and after applying DIREDUCER. The table shows that DIREDUCER is highly effective in anonymizing debug information. From the perspective of sizes, the average debug information size of the tested programs is 5.14 MB. After reduction, this decreases to merely 0.29 MB, achieving an approximately 95% reduction. From the perspective of attributes, on average, DIREDUCER reduces 94.55% of attributes in the DIG, and the corresponding source exposure metric $P(G^{'}, G)$ averages 5.45%, indicating that only a little debug information needs to be exposed for memory bug detection.

Breakdown of Debug Information Reduction. Figure 8 illustrates the respective contributions of selective pruning and type minimization. In the figure, each bar labeled by selective pruning represents the proportion of DIG attributes reduced during selective pruning, while a bar labeled by type minimization represents the proportion of additional DIG attributes reduced through type minimization. Selective pruning removes attributes such as `name`, `src_loc`, and `config`. Type minimization conducts deeper analysis by eliminating partial `type` and `mem_loc` attributes. On average, selective pruning removes 22.88% of DIG attributes, while type minimization removes 71.67%, achieving an overall reduction rate of 94.55%.

Breakdown of Compound Type Recovery. While type minimization ensures all types are theoretically recoverable at the granularity of primitive types, compound types (e.g., struct in C) may not be reassembled into their original composite forms, thus providing further protection of the source code. In Figure 9, **Retained** denotes the number of compound-type variables, whose types are preserved after type minimization. **Reduced** denotes the number of compound-type variables whose types are removed after type minimization and can be recovered only at the granularity of primitive types. On average, 40.5% of compound-type variables cannot be reassembled into their composite forms, showing that type minimization also provides a degree of source code protection.

B. RQ2 - Efficiency

Efficiency. Figure 11 shows that DIREDUCER's memory and time overhead exhibit a roughly linear growth relationship with the program sizes. The coefficient of determination, $R^2 \in [0, 1]$, measures the degree of fit between the data and the regression lines, where the value of R^2 closer to 1 indicates a better fit. For time overhead, the test results show R^2=0.8417, while for memory overhead it reaches R^2=0.9819.

For all test programs, DIREDUCER completes within a short time frame and with acceptable memory usage. Use the largest program, gcc, as an example. DIREDUCER takes only 35 seconds to process and only uses about 10GB of memory. These results demonstrate the graceful scalability of DIREDUCER in handling large-scale real-world programs.

Breakdown of Time Cost. In Figure 10, we break down the time overhead of DIREDUCER into four steps: Disassembly, Selective Pruning, Type Minimization, and Binary Reconstruction. For most programs, type minimization accounts for the most costly part, while selective pruning consumes the least. Additionally, the time overhead distribution across these four steps remains relatively consistent across most programs. However, we observed that for some programs (e.g., xz, nginx, and libiconv), the proportion of time spent on type minimization is significantly lower. Through debugging DIREDUCER, we observed that the time required for type minimization varies significantly based on function complexity. For large functions, i.e., those containing a higher number of variables and complex data structures, constructing the type dependency graph incurs higher overhead. In contrast, simpler functions with fewer variables and straightforward data structures require less time for type minimization.

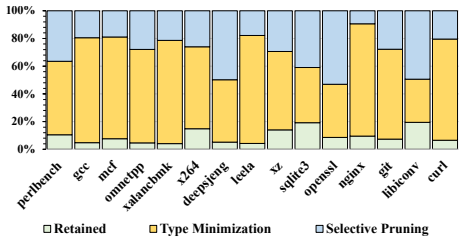


Fig. 8. Breakdown of Debug Information Reduction.

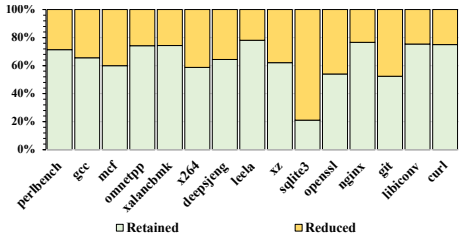


Fig. 9. Breakdown of Compound Type Reduction.

C. RQ3 - Sufficiency

While we have shown that DIREDUCER can reduce most of the source code information, the other goal of our work is to allow software vendors to justify that they have provided sufficient source code information for memory bug detection. In other words, we must show that, with proper techniques, the retained debug information (the type metadata and the memory locations) is sufficient for memory bug detection.

Evaluation Setup. To evaluate whether the retained debug information is sufficient for memory bug detection, we set up the experiments as follows. In the experiments, we use the static analyzers (saber [60] and ae [25]) from the SVF tool set [59] to check all common memory bugs it supports, including buffer overflow, memory leak, double free, etc. Since the static analyzers are LLVM-based, all source code and its binaries are compiled or decompiled into the LLVM IR before memory bug detection.

For the source code, we use WLLVM [12] to compile it into LLVM IR. For binaries, we employ Plankton [72], a state-of-the-art binary decompiler, to decompile them into LLVM IR. Three kinds of binaries are used in the experiment:

TABLE VI: Memory Bug Detection Results Compared with Full DI.

Program	Full DI	Partial DI			Without DI		
	#Report	#TP	#FN	#FP	#TP	#FN	#FP
perlbench	76	73	3	3	0	76	10
gcc	115	109	6	12	0	115	0
mcf	6	4	2	0	6	0	0
omnetpp	8	8	0	0	8	0	3
xalancbmk	0	0	0	0	0	0	0
x264	20	20	0	0	9	11	1
deepsjeng	0	0	0	0	0	0	0
leela	0	0	0	0	0	0	0
xz	4	4	0	0	1	3	0
sqlite3	5	5	0	0	2	3	2
openssl	0	0	0	0	0	0	0
nginx	7	5	2	2	1	6	0
git	274	261	13	5	0	274	0
libiconv	2	2	0	0	2	0	0
curl	4	4	0	0	4	0	0
Total	**521**	**517**			**49**		
	-	**495**	**26**	**22**	**33**	**488**	**16**
Rate	**-**	**95.0%**	**5.0%**	**4.2%**	**6.3%**	**93.7%**	**3.1%**

- Binaries with full debug information (**Full DI**).
- Binaries with all recoverable types and memory locations after being processed by DIREDUCER (**Partial DI**).
- Binaries without any debug information (**Without DI**).

It should be noted that while DIREDUCER significantly reduces the types and memory locations in DIG, this experiment uses binaries with all types and memory locations that can be recovered after being processed by DIREDUCER (i.e., Partial DI). The rationale of this design is explained as follows. When proving to a third party that software is free of memory bugs, software vendors can argue the sufficiency of debug information by claiming that all types and memory locations reduced by DIREDUCER are recoverable, as DIREDUCER only removes types that can be inferred from others, and after all type metadata and memory locations are recovered, it is sufficient for memory bug detection. Thus, in this experiment, we aim to show the sufficiency of Partial DI.

Comparing Partial DI to Full DI. Our objective is not to evaluate the performance of static analysis tools themselves but rather to assess the sufficiency of the debug information retained in binaries. Therefore, we use the bug reports produced by checking Full DI as the ground truth and compare the ground truth to the bugs reported on Partial DI and Without DI. True Positive (**TP**) denotes that a bug is detected in both Full DI and other binaries under comparison. False Negative (**FN**) denotes that a bug is reported in Full DI but is missing in others. False Positive (**FP**) denotes that a bug is not reported in Full DI but reported by others.

Table VI presents the bug detection results. The group with Full DI generates 521 bug reports, with git contributing over 200 reports. When using Partial DI, the TP rate reaches 95.0%, with 5.0% FN and 4.2% FP, demonstrating the sufficiency of the debug information retained by DIREDUCER. Through manual analysis of bug reports, we attribute the minor gap between Partial DI and Full DI to limitations in SVF's implementation, which occasionally causes SVF to fail when processing complex memory operations.

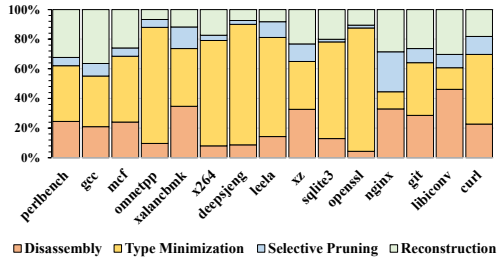


Fig. 10. Breakdown of Time Cost.

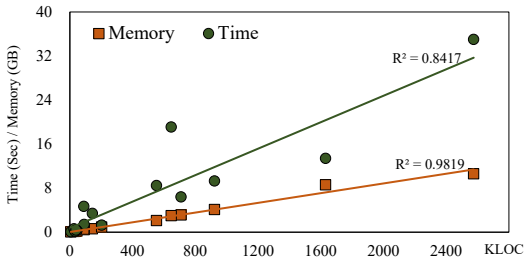


Fig. 11. Time and Memory Cost of DIREDUCER.

In contrast, without the debug information, the TP rate drops to only 6.3%, while the FN rate increases to 93.7%. This result shows that static analysis performs poorly on binaries without debug information, demonstrating the necessity of the debug information retained by DIREDUCER.

Comparing Partial DI to Source Code. We also compare the bug detection results produced by Partial DI and the source code, using the bug reports generated from the source code as ground truth. The definitions of TP, FN, and FP remain analogous to those previously stated, but the comparison baseline shifts from Full DI to the source code.

As shown in Table VII, static analysis on the source code, Full DI, and Partial DI yielded very similar numbers of bug reports, i.e., 515, 521, and 517 reports, respectively. For both Full DI and Partial DI, the TP rates are over 90%, with FP and FN rates less than 10%. In other words, using Partial DI (the binaries processed by DIReducer), the static analyzers produce comparable results, reporting over 90% of the reports with only a few FPs and FNs.

We manually examined the bug reports generated from the source code that were missed by Partial DI. On the one hand, all missed reports turned out to be false warnings (i.e., not genuine memory issues) that should not have been reported in the first place. On the other hand, the small discrepancies between the reports stem primarily from minor differences in the LLVM IR: Partial DI relies on the Plakton decompiler to produce LLVM IR, whereas the source-based analysis uses WLLVM. Although the two IRs are semantically equivalent and both provide sufficient type information for memory bug detection, subtle variations in instructions and control-flow structures can cause the bug detectors to behave slightly differently. For example, when a bitcast instruction is expressed as an equivalent getelementptr instruction, the analyzers may

TABLE VII: Memory Bug Detection Results Compared with Source Code.

Program	Source Code	Full DI			Partial DI		
	#Report	#TP	#FN	#FP	#TP	#FN	#FP
perlbench	74	73	1	3	73	1	3
gcc	124	108	16	7	106	18	15
mcf	4	4	0	2	4	0	0
omnetpp	11	8	3	0	8	3	0
xalancbmk	0	0	0	0	0	0	0
x264	22	20	2	0	20	2	0
deepsjeng	0	0	0	0	0	0	0
leela	0	0	0	0	0	0	0
xz	4	4	0	0	4	0	0
sqlite3	2	2	0	3	2	0	3
openssl	0	0	0	0	0	0	0
nginx	6	5	1	2	5	1	2
git	262	241	21	33	238	24	28
libiconv	2	2	0	0	2	0	0
curl	4	4	0	0	4	0	0
Total	**515**	**521**			**517**		
	-	**471**	**44**	**50**	**466**	**49**	**51**
Rate	**-**	**91.5%**	**8.5%**	**9.7%**	**90.5%**	**9.5%**	**9.9%**

diverge in their results. We regard such inconsistencies as implementation-specific issues of the static analyzers, which fall outside the scope of this paper.

In conclusion, static memory bug detectors exhibit comparable behavior when applied to source code, Partial DI, or Full DI. This observation is consistent with prior work [72], reinforcing that Partial DI provides sufficient information for effective memory bug detection.

VI. Discussion

Reverse Engineering Risks. Source code privacy is never absolute: even completely stripped binaries can be reverse engineered with sufficient manual effort. By retaining only minimal debug information (around 5%), DIReducer substantially increases the human effort required for reverse engineering compared to preserving full debug information. Meanwhile, this small portion of debug information is essential, as it enables us to demonstrate sufficiency for memory bug detection. We therefore believe that the trade-off in DIReducer i.e., balancing effective memory bug detection against reverse engineering risk, is both necessary and reasonable.

Necessity of DIReducer. First, as reported by the BBC [1], with growing demands for software security, vendors are often passively required (e.g., by governments) to provide source code for bug scanning, whereas vendors do not want to release their source code. Second, since deploying private and effective bug scanners is expensive, vendors (e.g., startups) may also actively want to send their code to a third party for bug scanning but still protect as much source code as possible. Thus, the industry needs such an approach that can justify the sufficiency for bug detection while protecting the source code.

Generality of DIReducer. DIReducer is currently implemented for the DWARF debug information format and x86-64 binaries. However, our methodology is applicable to other architectures and debug information formats. For debug information formats, DWARF is the de facto standard, and the Windows PDB format shares similar structural properties. For architectures, our methodology is architecture-agnostic and does not rely on any x86-64-specific features. Therefore, adapting DIReducer to other debug information formats or architectures does not involve higher theoretical barriers except for some engineering efforts.

VII. Related Work

Binary-Oriented Bug Detection. Bug detection in binaries is crucial in the fields of software quality assurance and often rely on static program analysis, dynamic program analysis, or a combination of both. BDA [20] and Bitblaze [57] are two well-known static analysis frameworks that provide extensive functionality for analyzing binaries. Pin [41] is a dynamic binary instrumentation framework that serves as the foundation for dynamic binary analysis. Over the years, a number of binary-oriented bug detection approaches have been developed to identify a variety of software vulnerabilities, such as numeric overflow [43], [55], [63], memory corruption [26], [27], [36], [45], [46], [49], [50], [56], [69], and uninitialized variable [31], to name a few. DIReducer is designed to detect these bugs and, meanwhile, protect the source code from being revealed.

Privacy-Preserving Program Analysis. Traditional program analysis techniques cannot be applied in certain sensitive scenarios due to the risk of source code leakage. Privacy-preserving program analysis techniques, however, enable program analysis while minimizing or entirely avoiding privacy disclosure. For example, Fang et al. [30] show the feasibility of privacy-preserving static analysis in a simplified language using zero-knowledge proof. However, due to the reliance on advanced cryptographic techniques, it typically suffers from significant performance overhead, making it challenging to deploy in real-world scenarios. Some other techniques [62], [64] also aim to preserve privacy but do not work for memory bugs. In contrast, DIReducer is inspired by conventional data anonymization techniques, which introduce minimal performance overhead and offer practical deployability in real-world scenarios for memory bug detection.

VIII. Conclusion

In this paper, we propose DIReducer, an approach to anonymizing debug information in binaries via selective pruning and type minimization. In situations where software vendors are required to demonstrate that their programs are free of memory bugs, DIReducer allows software vendors to reveal only a little (5%) source code information while simultaneously justifying that they have provided sufficient source code information for memory bug detection.

Acknowledgment

We extend our gratitude to the anonymous reviewers for their insightful comments and to Dr. Anshungkang Zhou for his assistance in setting up Plankton [72]. This work was partially supported by the NJU-HW Laboratory for Novel Software Technology (TC20230202021-2023-03).

REFERENCES

[1] BBC. https://www.bbc.com/news/business-20053511.

[2] Capstone - The Ultimate Disassembly Framework. https://www.capstone-engine.org/.

[3] Clang command line argument reference. https://clang.llvm.org/docs/ClangCommandLineReference.html.

[4] Cppcheck - A tool for static C/C++ code analysis. https://cppcheck.sourceforge.io/.

[5] DIReducer. https://github.com/Compiler-Security/DIReducer.

[6] DWARF debugging information format. https://dwarfstd.org/.

[7] Dwarf program information. https://www.ibm.com/docs/en/zos/2.4.0?topic=architecture-dwarf-program-information.

[8] GCC - Options for Debugging Your Program. https://cppcheck.sourceforge.io/.

[9] libdwarf. https://www.prevanders.net/dwarf.html.

[10] SPEC CPU 2017. https://www.spec.org/cpu2017/.

[11] The PDB (Program Database) Symbol File format. https://github.com/microsoft/microsoft-pdb.

[12] Wllvm: whole program llvm. https://github.com/travitch/whole-program-llvm.

[13] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. Ql: Object-oriented queries on relational data. In *Proceedings of the 30th European Conference on Object-Oriented Programming (ECOOP'16)*, 2016.

[14] Nathaniel Ayewah and William Pugh. Null dereference analysis in practice. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 65–72, 2010.

[15] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security (CCS'16)*, pages 356–367, 2016.

[16] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC'16)*, pages 189–200, 2016.

[17] Roberto J Bayardo and Rakesh Agrawal. Data privacy through optimal k-anonymization. In *Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 217–228, 2005.

[18] Irena Bojanova and Carlos Eduardo Galhardo. Classifying memory bugs using bugs framework approach. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC'21)*, pages 1157–1164, 2021.

[19] Michael D Brown, Matthew Pruett, Robert Bigelow, Girish Mururu, and Santosh Pande. Not so fast: understanding and mitigating negative impacts of compiler optimizations on code reuse gadget sets. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, 2021.

[20] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*, pages 463–469, 2011.

[21] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 21st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'12)*, pages 133–143, 2012.

[22] Juan Caballero, Noah M Johnson, Stephen McCamant, and Dawn Song. Binary code extraction and interface identification for security applications. In *Network and Distributed Systems Security Symposium (NDSS'10)*, 2010.

[23] Mariano Ceccato, Massimiliano Di Penta, Jasvir Nagra, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. The effectiveness of source code obfuscation: An experimental assessment. In *2009 IEEE 17th International Conference on Program Comprehension (ICPC'09)*, pages 178–187, 2009.

[24] 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 *31st USENIX Security Symposium (USENIX Security'22)*, pages 4327–4343, 2022.

[25] Xiao Cheng, Jiawei Ren, and Yulei Sui. Fast graph simplification for path-sensitive typestate analysis through tempo-spatial multi-point slicing. In *Proceedings of the 32nd ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*, pages 494–516, 2024.

[26] Tobias Cloosters, Michael Rodler, and Lucas Davi. Teerex: Discovery and exploitation of memory corruption vulnerabilities in sgx enclaves. In *29th USENIX Security Symposium (USENIX Security'20)*, pages 841–858, 2020.

[27] David Dewey and Jonathon T Giffin. Static detection of c++ vtable escape vulnerabilities in binary code. In *Network and Distributed Systems Security Symposium (NDSS'12)*, 2012.

[28] Yufei Du, Omar Alrawi, Kevin Snow, Manos Antonakakis, and Fabian Monrose. Improving security tasks using compiler provenance information recovered at the binary-level. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS'23)*, pages 2695–2709, 2023.

[29] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. Scalable variable and data type detection in a binary rewriter. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation (PLDI'13)*, pages 51–60, 2013.

[30] Zhiyong Fang, David Darais, Joseph P Near, and Yupeng Zhang. Zero knowledge static program analysis. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'21)*, pages 2951–2967, 2021.

[31] Behrad Garmany, Martin Stoffel, Robert Gawlik, and Thorsten Holz. Static detection of uninitialized stack variables in binary code. In *European Symposium on Research in Computer Security (ESORICS'19)*, pages 68–87, 2019.

[32] Aristides Gionis and Tamir Tassa. k-anonymization with minimal loss of information. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 21(2):206–219, 2008.

[33] Binfa Gui, Wei Song, Hailong Xiong, and Jeff Huang. Automated use-after-free detection and exploit mitigation: How far have we gone? *IEEE Transactions on Software Engineering (TSE)*, 48(11):4569–4589, 2021.

[34] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. Typesan: Practical type confusion detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*, pages 517–528, 2016.

[35] 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)*, pages 1667–1680, 2018.

[36] Liang He, Yan Cai, Hong Hu, Purui Su, Zhenkai Liang, Yi Yang, Huafeng Huang, Jia Yan, Xiangkun Jia, and Dengguo Feng. Automatically assessing crashes from heap overflows. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*, pages 274–279, 2017.

[37] David S Johnson. Approximation algorithms for combinatorial problems. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 38–49, 1973.

[38] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-llvm–software protection for the masses. In *2015 ieee/acm 1st international workshop on software protection*, pages 3–9, 2015.

[39] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. Type casting verification: Stopping an emerging attack vector. In *24th USENIX Security Symposium (USENIX Security'15)*, pages 81–96, 2015.

[40] JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. In *Network and Distributed Systems Security Symposium (NDSS'11)*, 2011.

[41] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. page 190–200, 2005.

[42] Xutong Ma, Jiwei Yan, Wei Wang, Jun Yan, Jian Zhang, and Zongyan Qiu. Detecting memory-related bugs by tracking heap memory management of c++ smart pointers. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'21)*, pages 880–891, 2021.

[43] David Molnar, Xue Cong Li, and David A Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th Conference on USENIX Security Symposium (USENIX Security'09)*, pages 67–82, 2009.

[44] Suntherasvaran Murthy, Asmidar Abu Bakar, Fiza Abdul Rahim, and Ramona Ramli. A comparative study of data anonymization techniques.

In *2019 IEEE 5th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing (HPSC), and IEEE Intl Conference on Intelligent Data and Security (IDS)*, pages 306–309, 2019.

[45] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. Binary-level directed fuzzing for use-after-free vulnerabilities. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 47–62, 2020.

[46] Bindu Madhavi Padmanabhuni and Hee Beng Kuan Tan. Buffer overflow vulnerability prediction from x86 executables using static analysis and machine learning. In *Proceedings of the 2015 IEEE 39th Annual Computer Software and Applications Conference (COMPSAC'15)*, pages 450–459, 2015.

[47] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE symposium on security and privacy (SP'21)*, pages 833–851, 2021.

[48] 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'21)*, pages 690–702, 2021.

[49] Aravind Prakash, Xunchao Hu, and Heng Yin. vfguard: Strict protection for virtual function calls in cots c++ binaries. In *Network and Distributed Systems Security Symposium (NDSS'15)*, 2015.

[50] Manish Prasad and Tzi-cker Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *Proceedings of the 2003 USENIX Conference on Annual Technical Conference (USENIX ATC'03)*, pages 211–224, 2003.

[51] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *Acm computing surveys (CSUR)*, 49(1):1–37, 2016.

[52] 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. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*, pages 693–706, 2018.

[53] Qingkai Shi, Xiaoheng Xie, Xianjin Fu, Peng Di, Huawei Li, Ang Zhou, and Gang Fan. Datalog-based language-agnostic change impact analysis for microservices. In *Proceedings of the International Conference on Software Engineering (ICSE'25)*, pages 652–652, 2025.

[54] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE symposium on security and privacy (SP'16)*, pages 138–157, 2016.

[55] Stelios Sidiroglou-Douskos, Eric Lahtinen, Nathan Rittenhouse, Paolo Piselli, Fan Long, Deokhwan Kim, and Martin Rinard. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*, pages 473–486, 2015.

[56] Asia Slowinski, Traian Stancescu, and Herbert Bos. Body armor for binaries: preventing buffer overflows without recompilation. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*, pages 125–137, 2012.

[57] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS'08)*, pages 1–25, 2008.

[58] Venkatesh Srinivasan and Thomas Reps. Recovery of class hierarchies and composition relationships from machine code. In *International Conference on Compiler Construction (CC'14)*, pages 61–84, 2014.

[59] Yulei Sui and Jingling Xue. Svf: Interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction (CC'16)*, pages 265–266, 2016.

[60] Yulei Sui, Ding Ye, and Jingling Xue. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA'12)*, pages 254–264, 2012.

[61] David A Tomassi and Cindy Rubio-González. On the real-world effectiveness of static bug detectors at finding null pointer exceptions. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'21)*, pages 292–303, 2021.

[62] Huaijin Wang, Zhibo Liu, Yanbo Dai, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. Preserving privacy in software composition analysis: A study of technical solutions and enhancements. In *Proceedings of the 47th International Conference on Software Engineering (ICSE'25)*, pages 592–592, 2025.

[63] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Network and Distributed Systems Security Symposium (NDSS'09)*, 2009.

[64] Zhaoyu Wang, Pingchuan Ma, Huaijin Wang, and Shuai Wang. Pp-csa: Practical privacy-preserving software call stack analysis. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):1264–1293, 2024.

[65] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proceedings of the International Conference on Software Engineering (ICSE'18)*, pages 327–337, 2018.

[66] Chengfeng Ye, Yuandao Cai, Anshunkang Zhou, Heqing HUANG, Hao Ling, and Charles Zhang. Manta: Hybrid-sensitive type inference toward type-assisted bug detection for stripped binaries. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24)*, 2024.

[67] Kyungjin Yoo and Rajeev Barua. Recovery of object oriented features from c++ binaries. In *2014 21st Asia-Pacific Software Engineering Conference*, pages 231–238, 2014.

[68] Yizhuo Zhai, Zhiyun Qian, Chengyu Song, Manu Sridharan, Trent Jaeger, Paul Yu, and Srikanth V Krishnamurthy. Don't waste my efforts: Pruning redundant sanitizer checks by {Developer-Implemented} type checks. In *33rd USENIX Security Symposium (USENIX Security'24)*, pages 1419–1434, 2024.

[69] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. Vtint: Protecting virtual function tables' integrity. In *Network and Distributed Systems Security Symposium (NDSS'15)*, 2015.

[70] Haoxiang Zhang, Shaowei Wang, Heng Li, Tse-Hsun Chen, and Ahmed E Hassan. A study of c/c++ code weaknesses on stack overflow. *IEEE Transactions on Software Engineering (TSE)*, 48(7):2359–2375, 2021.

[71] 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'21)*, pages 813–832. IEEE, 2021.

[72] Anshunkang Zhou, Chengfeng Ye, Heqing Huang, Yuandao Cai, and Charles Zhang. Plankton: Reconciling binary code and debug information. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24)*, pages 912–928, 2024.