Sdft: A PDG-based Summarization for Efficient Dynamic Data Flow Tracking

Xiao Kan
Xidian University
Xi’an, China
814091656@qq.com

Cong Sun†
Xidian University
Xi’an, China
suncong@xidian.edu.cn

Shen Liu
NVIDIA
Santa Clara, CA, USA
sheliu@nvidia.com

Gang Tan
The Pennsylvania State University
University Park, PA, USA
gtan@psu.edu

Siqi Ma
The University of Queensland
Brisbane, Australia
slivia.ma@uq.edu.au

Yongzhe Huang
University Park, PA, USA
yzh89@psu.edu

Yumei Zhang
Xidian University
Xi’an, China
zhangyumei319@163.com

Abstract—Dynamic taint analysis (DTA) has been widely used in various security-relevant scenarios that need to track the runtime information flow of programs. Dynamic binary instrumentation (DBI) is a prevalent technique in achieving effective dynamic taint tracking on commodity hardware and systems. However, the significant performance overhead incurred by dynamic taint analysis restricts its usage in production systems. Previous efforts on mitigating the performance penalty fall into two categories, parallelizing taint tracking from program execution and abstracting the tainting logic to a higher granularity. Both approaches have only met with limited success.

In this work, we propose Sdft, an efficient approach that combines the precision of DBI-based instruction-level taint tracking and the efficiency of function-level abstract taint propagation. First, we build the library function summaries automatically with reachability analysis on the program dependency graph (PDG) to specify the control- and data dependencies between the input parameters, output parameters, and global variables of the target library. Then we derive the taint rules for the target library functions and develop taint tracking for library function that is tightly integrated into the state-of-the-art DTA framework Libdft. By applying our approach to the core C library functions of glibc, we report an average of 1.58x speed up over the target program.

Index Terms—dynamic taint analysis, dynamic binary instrumentation, information flow, program dependency graph

I. INTRODUCTION

Dynamic taint analysis (DTA), also known as dynamic data-flow tracking (DFT), is a powerful technique for tracking information flows of software at runtime and has been used widely in vulnerability detection, program protection, information flow control, and reverse engineering. Without accessing the source code, binary-level dynamic data-flow tracking usually uses some runtime techniques, e.g., dynamic binary instrumentation (DBI), virtual machine monitor, or whole system emulator, to monitor the target program transparently and propagate sensitive taints across memory and program contexts. Then the knowledge on taint propagation is collected directly over the target binary.

The DBI-based dynamic taint analyses [1]–[5] are promising and flexible to track in-process tainting behaviors. Such approaches mainly focus on direct data flows and hold the tainting states within tagging memory. The key feature is to track memory locations and CPU registers that store sensitive or suspicious data. This kind of tainted data is propagated and checked at particular program execution points to decide if specific runtime properties are satisfied, e.g., whether some pointer in instruction is controlled and tampered with by attackers. To generalize the DBI-based approaches, several extensions have addressed implicit data flows [6], flows among multiple processes [7], or the generalization of taint propagation semantics to more instruction set architectures (ISA) [8]. Libdft [5], [9] and its 64-bit reimplementation [10] are the leading DBI-based taint tracking approach that has a relatively moderate slowdown to the native execution [11], [12]. This efficient dynamic taint analysis has been used to capture the data provenance [13] or the common characteristics of valid inputs of gray-box fuzzing [14], [15].

The significant performance penalty of dynamic taint analysis has been a prominent weakness for a long time. The complex taint tracking takes a much longer time to execute the instrumented program than the original program. Improvements are on two orthogonal dimensions, i.e., parallelization or sequential abstraction. The data tracking can be offloaded/decoupled from the program execution to introduce more parallelization over different cluster nodes, CPUs, hosts, processes, or threads [16]–[20]. On the other hand, we can aggregate the taint analysis from per-instruction tracking to a higher granularity. The tainting logic of code segments is specified at a more abstract level, e.g., at each basic block or function. At the basic block level, the tainting operations are checked at particular program execution points to decide if specific runtime properties are satisfied, e.g., whether some pointer in instruction is controlled and tampered with by attackers. To generalize the DBI-based approaches, several extensions have addressed implicit data flows [6], flows among multiple processes [7], or the generalization of taint propagation semantics to more instruction set architectures (ISA) [8]. Libdft [5], [9] and its 64-bit reimplementation [10] are the leading DBI-based taint tracking approach that has a relatively moderate slowdown to the native execution [11], [12]. This efficient dynamic taint analysis has been used to capture the data provenance [13] or the common characteristics of valid inputs of gray-box fuzzing [14], [15].
mechanism at specific program points where the live locations are untainted to support switching to an efficient version of code without taint tracking computations. At the function level, TaintEraser [22] first proposes function-level summaries for Windows kernel APIs to improve the efficiency of dynamic taint tracking. The function summaries of this approach are specified on-demand by human effort, thus preventing the usage on a large scale to the standard libraries. Automatically inferring the library summaries for the information flows has been proposed in different application scenarios, e.g., Android libraries [23], [24]. Static reachability analysis on data-flow graphs has been used to identify data propagation paths [11]. The derived paths have been neither used to derive function summaries nor integrated into any dynamic taint analysis. By observing that the function-level taint tracking is more abstract and efficient than the instruction-level taint tracking, the work in this paper belongs to the category of sequential abstraction.

In this paper, we present Sdft, a framework that automatically derives library function summaries and taint rules to improve the efficiency of dynamic taint analysis. Sdft analyzes the library source code to generate interprocedural program dependency graphs (PDG) for the target library functions. On the PDGs, we use reachability analysis to derive the function summaries specifying the control- and data dependencies between the input parameters, the output parameters, and the global variables of the library. Then we derive the function-level taint rules for the target library functions. Because the PDG-based analysis requires source code, to address the usability, we focus on the abstraction of standard library functions whose source code is obtainable. We apply our approach to the core functions of the standard C library, i.e., glibc. We implement a dynamic binary instrumentation tool by extending the state-of-the-art dynamic taint analysis framework Libdft [5], [10]. Our dynamic taint analysis tool can switch at runtime between the instruction-level user-code taint tracking and the function-level library function taint tracking. The contributions of this paper are summarized as follows:

1) We propose a PDG-based automatic function summarization and taint rule generation for modeling the tainting behaviors of library functions invoked by the applications.
2) We tightly integrate the function-level tainting behavior abstractions into the dynamic taint analysis by developing an extension of the state-of-the-art DTA framework Libdft.
3) We apply our approach on glibc 2.27 and evaluate Sdft on the efficiency, tainting effects, and effectiveness of vulnerability tracking. By abstracting the core functions of the standard C library, Sdft can achieve an average 1.58x speed up on performance compared with Libdft64.

II. MOTIVATING EXAMPLE

We present the motivating example in Fig. 1. In the example, a simplified memcpy is used by another function student_cpy to copy a struct to a global struct of student. We assume both memcpy and student_cpy are library functions wrapped in a shared object libcopy.so.

```c
// libcopy.so :
typedef struct {
  char id[SIZE];
  int score;
} student;
student stu;

void *memcpy(void *dest, const void *src, size_t n) {
  char *dp = dest;
  const char *sp = src;
  while (n--) {
    *dp++ = *sp++;
    return dest;
  }
}

void student_cpy (student *src) {
  memcpy(stu.id, src->id, SIZE);
  stu.score = src->score;
}

// main.c :
int main(){
  student s;
  fgets(s.id, SIZE-1, stdin);
  s.score = 95;
  student_cpy(&s);
  printf("(%s: %d)\n", stu.id, stu.score);
}
```

Fig. 1. Application of a simplified memcpy

To facilitate data-flow tracking, we assume the standard I/O function fgets as data source and printf as data sink. In the user code main.c, the critical input from fgets is copied by the library function student_cpy to the global student struct and finally printed by printf. Consequently, a sensitive flow should be captured by our dynamic taint analysis approach. The state-of-the-art instruction-level taint analysis, e.g., Libdft [5], [10], will go through both the user code (i.e., the binary of main) and the library code (i.e., the binary of libcopy.so) to conduct per-instruction instrumentation and taint propagation. In contrast, we will elaborate that our approach generates effective function summary and taint rules for the library functions (i.e., memcpy and student_cpy) and applies the rules in the dynamic taint analysis framework to avoid the instructions in these functions being instrumented, thus improving the overall efficiency.

III. DESIGN OF SDFT

In this section, we describe the framework of Sdft, especially on generating the summaries of functions and their taint rules used in the dynamic taint tracking of the framework.

A. System Overview

Key Concepts of Libdft. As the state-of-the-art DBI-based dynamic taint analysis framework, Libdft [5] and its descendant Libdft64 [10] are Pintools developed on top of the Intel DBI framework Pin [25]. Libdft has three main components: Tagmap, tracker, and I/O Interface, as shown in Fig. 2. The Tagmap provides memory space and operation interface of the shadow memory and shadow registers, which models the runtime tainting state of the program. The tracker instruments the binary program with proper analysis routines before each
consDecl ::= struct(id, φ) | union(id, φ) (Field list)

libStmt ::= extern τ gvar; (Gvar Decl)

funStmt ::= function(τ0, ..., τn−1, τn) id; (Func Decl)

H ::= {consDecl | libStmt | funStmt}+ (Headers)

Fig. 3. Abstract syntax of C language header

Algorithm 1 Flatten Complex Types to Primitive Types

1: procedure FLATTENPRIMITYPES(τ)
2:  types ← ∅, primTypeMap ← ∅
3:  for all cons(id, φ) ∈ C do
4:      COLLECTPRIMITYPE(cons(id, φ), types)
5:      primTypeMap.put(id ↔ types)
6:  return primTypeMap
7: procedure COLLECTPRIMITYPE(cons(id, φ), typeSet)
8:  Suppose φ ≡ {(id1, τ1), ..., (idk, τk)}
9:  for all τ ∈ {τ1, ..., τk} do
10:     if isPrimType(τ) then
11:        typeSet.add(τ)
12:     else if τ ≡ cons(id, φ′) then
13:        COLLECTPRIMITYPE(τ, typeSet)

Fig. 2. Framework of Sdft

Libdft backend
Tagmap
I/O Interface
(modified)
Tracker
(modified)
Library 
source code
PDG Lib Function 
Summaries
Taint Rules of 
Lib Functions
DFT results
Offline phase
Online phase
App Binary
I/O Interface
(modified)
Libdft backend

Instruction to operate on the taint tags according to the data-flow tracking logic of each instruction. The I/O Interface handles the taint propagation and sanitization of system calls. For each system call, this component conducts a pre/post-call-site instrumentation that inserts stubs running in user mode.

The Framework of Sdft. As presented in Fig. 2, the framework of Sdft consists of two phases: the offline phase and the online phase. The offline phase generates the taint rules from the source code of library functions. In this phase, we firstly use an off-the-shelf PDG generator [25] to generate an interprocedural PDG for library functions. We parse the library headers in source code to flatten complex data types (Section III-B) and then generate the function summaries with reachability analysis (Section III-C). Then, we derive the taint rules of library functions (Section III-D) as the critical configuration of the online data-flow tracking. On the other hand, the online phase launches the execution of binary and conducts dynamic data-flow tracking. For this phase, we propose an extension on Libdft64 to support an instruction-level and function-level interleaving data-flow tracking. The extension includes a new module of taint tracker and modifications on the main modules of Libdft64, as demonstrated in Section IV.

B. Library headers parsing

We take a similar abstract syntax of the C language types as in [27]. The abstract syntax of types and the library headers are in Fig. 3. For the type alias in the library defined as typedef(τ, τ), we assume the alias τ has been eliminated by static substitution. For a specific library ℓ, the library headers Hℓ consist of a set of struct or union declarations Cℓ, a list of global variable definitions Gℓ, and a list of API declarations Fℓ. In our static function summary generation, unions are treated conservatively in the same way as structs. Therefore in the following, we also use cons(id, φ) to stand for struct(id, φ) or union(id, φ). Treating union in the same way as struct will not bring in imprecision because in the well-formed library definition, even we enumerate all the members of a union as the input or output of a function, only one member will be used in the tainting procedure at runtime. The rest members will not receive or propagate taints and their taint rules will not be used. We use Algorithm 1 to flatten complex types (unions and structs) to a set of primitive types. In the algorithm, isPrimType(τ) decides if the type τ is int, float, char, void, or the pointer of these types. The primitive types of each struct or union can be retrieved from primTypeMap with the id of struct or union.

C. PDG-based Summary Generation

The summary of a function specifies the control- and data-dependencies between the input parameters, the output parameters, and the global variables of the library. Such relations are built with a PDG. The interprocedural PDG of each library function is derived using PtrSplit [26]. In the PDG of each function, we choose proper nodes and apply reachability analysis to derive the summaries. Firstly, we define the following predicates that operate on the PDG:

1) findPath(n_s, n_t): Depth-first traversal from node n_s to n_t in PDG, returning all the paths consisting of the edges with type D_gnrl, DEF_USE, or RAW, as defined in Table VIII.
2) findNode(ins, p): For the LLVM-IR instruction ins in function p, the predicate gets the node of ins in the PDG of function p.
3) findNextUse(ins): Returning the target node of a DEF_USE edge, if the LLVM-IR instruction ins is in the source node of this edge. If ins assigns a value to variable v, the predicate returns the nearest subsequent instruction in the current function that uses v.

Intuitively, an edge (n_j, n_i) in a path found by findPath indicates n_j is data-dependent to n_i. The path returned by
fetchPath specifies a relationship between the source and target node s.t. when the source node defines variable/parameter, computes a new value, or reads memory locations, how the data involved propagate, and how they are interfered with by the taint data in the execution of the function. We do not consider the data dependency edges with type D_ALIAS since they may trigger many irrelevant circular paths. To figure out the implicit flows in some library functions as mentioned in Section \ref{sec:library} we also integrate the control dependency edge identification as an option of fetchPath.

For the reachability analysis, we first decide the source nodes and target nodes for the PDG of each library function. The source and target node respectively represent the input and output of each library function. For a library function \( p \). Let \( N^s_p \) and \( N^t_p \) be the set of source and target nodes in its PDG. To derive \( N^s_p \), we first collect the candidate node of parameters/global variables. Then we add the primitive-type candidates into \( N^s_p \). For the struct or union candidate, if some of its fields are used, i.e., loaded by some load instruction under a specific instruction sequence pattern, we add the node of loading action into \( N^s_p \). To derive \( N^t_p \), we first label the return instruction of function \( p \) as a target node in \( N^t_p \). Then, if we find some primitive-type global variable used in \( p \) and stored with some value, such store instruction serves as an output of \( p \). For the pointer-type global variables and parameters, if the memory pointed by these variables or parameters is modified by some store instruction in the function under specific instruction sequence patterns, the store instruction is added to \( N^t_p \).

The summaries are defined on the parameters and return value of each function and the global variables. However, the nodes captured in \( N^s_p \) and \( N^t_p \) cannot always stand for the function parameter, global variable, or return instruction. We define a mapping relation \( \phi \) from the source or target nodes to the function parameters, global variables, and return instructions, i.e. \( \phi : N^s_p \cup N^t_p \rightarrow N_{para} \cup N_{glb} \cup N_{ret} \), s.t. \( N_{para}, N_{glb}, \text{ and } N_{ret} \) are defined in Table \ref{tab:summary}. \( \phi(n) \) represents the parameter, global variable, or return instruction bound to the PDG node \( n \). In reality, \( \phi \) is complicated to support fine-grained relations from the source/target nodes to some field of struct parameters. We take ad-hoc instruction pattern analysis to figure out these relations.

After collecting the source and target nodes in the PDG of each library function, we compute the data- and control dependencies between the source and target nodes with reachability analysis. For each source node \( n_s \in N^s_p \) and each target node \( n_t \in N^t_p \) if we find a path from \( n_s \) to \( n_t \), we construct the summary relation over \( \phi(n_s) \) and \( \phi(n_t) \). For clarity to specify the summaries, we aggregate the inputs that each output depends on, i.e. \( \text{summaries} := \{\{n^{out}_{i}, \{n^{in}_{i1}, \ldots, n^{in}_{ik_i}\}\}_{i=1..m}\} \). For the function memcpy in Fig. \ref{fig:memcpy} the generated summaries are in Fig. \ref{fig:memcpy_summary}. The summaries begin with the input and output parameters and global variables captured by \( \phi \) on the source and target nodes. The id of the PDG node is attached to each parameter and variable. The specific paths with the same source and target node are organized together following the source-target pair and are omitted in Fig. \ref{fig:memcpy_summary} for simplicity. For the summary of student_cpy, the procedure fetchPath takes the summary of the callee function memcpy as input to build its dependencies. This modular treatment makes the procedure of summary generation efficient compared with the global PDG traversal.

D. Taint Rule Generation

The back-end dynamic tainting engine usually holds shadow memory and shadow registers for tracking the tainting behavior of the program. When a specific memory space or register is tainted, the tag value of the respective shadow memory or shadow register is set. To specify the tainting behaviors of function for our approach, the taint rules of each function are a kind of relation between the elements of shadow memory and shadow registers. More specifically, this relation can be modeled as \( R_p = (\langle \mu, \nu \rangle, \langle \mu', \nu' \rangle) \). \( \mu \) and \( \mu' \) are respectively the state of shadow memory before and after the execution of function \( p \). \( \nu \) and \( \nu' \) are the state of shadow registers before and after the execution of \( p \). For a library function, we map the elements of \( \langle \mu, \nu \rangle \) to the function input and map the function output to the elements of \( \langle \mu', \nu' \rangle \). Then we know through this function, the possible taints can impact which part of shadow memory or registers. The taint rules are derived to operate on the involved taint tags.

We assume the tagging granularity is at the byte level. For the memory region accessed by the library function, we present two predicates to facilitate the modeling of \( R_p \):

1. \text{getTaint(addr, sz)}: For the \( sz \)-byte memory region at \( addr \), union the respective 1-byte tags in \( \mu \) iteratively, and return the result tag byte.

2. \text{setTaint(addr, tag, sz)}: For the \( sz \)-byte memory region at \( addr \), set each tag byte in the respective tag region of \( \mu \) with \( tag \).

Similarly, \( addr \) in these primitives can be replaced by the \text{id} of the general-purpose registers. Then the predicates can also work on the shadow registers in \( \nu \). In these definitions, the tag granularity is in byte. Without loss of generality, our framework can also support bit-level taint tracking with a different definition of getTaint and setTaint.
Algorithm 2 Taint Rules Generation of Function p

procedure TaintRuleGen(summaries,p)

rules ← ∅
for all \((n^{out}_i,(n^1,...,n^k)) \in \text{summaries}_p\), do
\(tag_{out} \leftarrow 0\)
for all \(n^{out}_i \in \{n^1,...,n^k\}\) do
if \(\text{isPrimType}(T(n^{out}_i)) \land T(n^{out}_i) \notin \{\text{char*},\text{void*}\}\) then
rules.append('tag_{out} \leftarrow \text{getTaint}(A(n^{out}_i),\text{sizeof}(T(n^{out}_i)))
else if \(T(n^{out}_i) \in \{\text{char*},\text{void*}\}\) then
rules.append('tag_{out} \leftarrow \text{getTaint}(A(n^{out}_i),\text{sizeof}(T(n^{out}_i)))
endif
if \(\text{isPrimType}(T(n^{out}_i)) \land T(n^{out}_i) \notin \{\text{char*},\text{void*}\}\) then
rules.append('setTaint(A(n^{out}_i),tag_{out},\text{sizeof}(T(n^{out}_i)))
else if \(T(n^{out}_i) \in \{\text{char*},\text{void*}\}\) then
rules.append('setTaint(A(n^{out}_i),tag_{out},\text{sizeof}(T(n^{out}_i)))
endif
endfor
return rules

IV. IMPLEMENTATION ISSUES

Sdft provides function-level tainting for library functions to improve the efficiency of the instruction-level tainting of Libdft. As shown in Fig. 2, we develop a LibTaintTracker component to instrument the library functions whose taint rules are generated by the offline phase (summary generation and taint rule generation). Differently from the pre/post-syscall instrumentation by I/O interface, LibTaintTracker performs callee-side instrumentation at the return points of the library function to insert the taint rules generated by Algorithm 2 as a stub. Besides, LibTaintTracker can specify a library function as a taint source or sink on demand. To the taint source function, it uses the operations of Tagmap to introduce sensitive tags into Tagmap. To the taint sink function, it instruments vulnerability checking code at the beginning of the function.

To implement correct switching between function-level tainting and instruction-level tainting, the instructions of the library functions should be identified to avoid being instrumented at the instruction level. To distinguish the user-code instructions from the instructions inside the library functions, we define a global flag to reserve if the current context is inside or outside library functions. We modify the module tracker of Libdft to take the value of this flag to decide whether to instrument the instructions of the current basic block. Besides, we also modify the I/O Interface component to avoid instrumenting the system calls invoked inside the library functions. In our implementation, I/O Interface contains the abstraction of 335 system calls of Linux kernel v5.4.0-72.

To be more specific on generating the taint rules with Algorithm 2 the predicates getTaint and setTaint are mapped to the operation API of Tagmap, and the taint rules are mapped to the sequences of Tagmap operations. The implementation of predicate \(A()\) depends on the application binary interface (ABI) of the binaries under instrumentation. In this work, the target ISA of Sdft is x64 and the ABI we use is System V ABI. However, the taint rules generated by Algorithm 2 are ABI-neutral. The calling convention decides how \(A()\) maps the PDG node to the memory locations or the registers. For
example, the nodes of the first six integer parameters are mapped to the general-purpose register rdi, rsi, rdx, rcx, r8, r9. The nodes of the SSE parameters are mapped to xmm0 to xmm7. For the global variables, the DBI framework cannot provide sufficient addressing information for A(). We find the offset of each global variable statically in the shared object file of the library. The address of the global variable can be derived by adding this offset to the base address of the library returned by the DBI framework with Pin’s IMG_LowAddress().

For the variadic library functions, several arguments cannot directly map to the parameter nodes of the function’s PDG. Our instrumented code snippet figures out the number of arguments and provides parameterized and iterative taint rules that take the number of arguments as a parameter. We mainly focus on the variadic functions in stdio.h of the C standard library. For these functions, we take heuristics to capture the number and type of arguments by analyzing the format strings in the binary. Then we iteratively get the stack address of these parameters using Pin’s API.

V. EVALUATION

Our experiments are conducted on a Desktop with a 3.2GHz×4 Intel Core(TM) i5-6500 CPU, 8GB RAM, Linux 5.4.0-72 kernel (Ubuntu 18.04 64-bit). The binaries are compiled with GCC v7.5.0. The PDG [26], [28]–[30] depends on LLVM v9.0.0. The DBI framework is Pin v3.11 and Libdft64 we used is commit id 729c1b2 [10].

The benchmark programs we use to evaluate the performance and effectiveness of Sdft are presented in Table I. We compile the binaries on their default compiler optimization level. For each program, we feed the execution of the instrumented program with specific inputs to conduct the following evaluations. Specifically, for the SPEC2k6 benchmarks, we use the standard workload test in the instrumented execution. For the other benchmarks, we enumerate the command options and launch multiple instrumented executions in a batch job using different options. For each command option, we only feed with several common arguments. This setting will not trigger a high code coverage of each benchmark but is proper to demonstrate the performance of different approaches under routine usages. We count the number of instructions in the static binaries, i.e., #Instr in Table I. We also profile the code coverage of each benchmark under the instrumented executions, i.e.,

\[
\frac{\#\text{BBL}_{\text{exec}}}{\#\text{BBL}_{\text{total}}} \times 100\% \quad (1)
\]

where \#BBL_{exec} is the number of basic blocks in the CFG reached by the execution under the specific inputs and \#BBL_{total} is the total number of basic blocks in the CFG of binary. The code coverage of benchmark programs ranges between 11.6% to 64.1%.

A. Instantiated instrumentation

Our instrumentation has been applied to a subset of library functions of glibc v2.27. The main reason that we summarize the library functions instead of the user functions is that our PDG-based analysis requires source code and the source code of library functions is far easier to be obtained compared with the source code of the user program. Indeed, the shared object file libc.so of glibc 2.27 has 1833 library functions. Instrumenting all of these functions with Sdft will cause a very long loading time of libc.so in the execution of the instrumented target program. More practically, we choose to instrument a major subset of the standard C library functions, which include 98 standard C library functions. All of them are declared in stdio.h, stdlib.h, string.h, or time.h.

We evaluate this instantiated instrumentation from the following aspects. Firstly, to demonstrate the representativeness of the 98 candidate functions, we investigate the proportion of calls to the candidate functions in the calls to all the glibc functions in the batch-job executions of the benchmark programs. Table II presents the proportions. We can see the 98 candidate functions take 56.5–95.1% calls to the libc.so. These functions have not dominated the calls to libc.so in

| TABLE I | BENCHMARKS AND CODE COVERAGE UNDER TYPICAL INPUTS |
|---------|----------------------------------|
| Program (version) | #Instr | Code Coverage(%) |
| 400.perbench(SPEC2k6) | 305,050 | 32.3 |
| 401.bzip2(SPEC2k6) | 23,466 | 63.4 |
| 403.gcc(SPEC2k6) | 916,519 | 31.2 |
| thttpd(2.25) | 12,499 | 17.9 |
| wget(1.19) | 66,974 | 14.5 |
| nginx(1.9.5) | 175,294 | 11.6 |
| MySQL(5.7.33) | 111,743 | 21.7 |
| FireFox(4.0) | 10,481 | 64.1 |

| TABLE II | REPRESENTATIVENESS OF INSTRUMENTED LIBRARY FUNCTIONS |
|-----------|----------------------------------|
| Program | #calls functions (×10^5) | #calls candidate functions (×10^7) | (%) |
| 400.perbench | 2.54 | 2.22 | 87.5 |
| 401.bzip2 | 1.60 | 1.44 | 90.2 |
| 403.gcc | 2.36 | 2.08 | 88.3 |
| thttpd | 10.16 | 5.74 | 56.5 |
| wget | 4.02 | 1.41 | 70.1 |
| nginx | 25.62 | 23.77 | 92.7 |
| MySQL | 15.16 | 14.42 | 95.1 |
| FireFox | 0.22 | 0.20 | 90.8 |

| TABLE III | NUMBER OF SUMMARIES AND TAINT RULES OF STANDARD C LIBRARY FUNCTIONS |
|------------|----------------------------------|
| Header file | #func | Avg. #summary per func | Avg. #rule per func |
| stdio | 41 | 2.00 | 0.27 | 0.17 | 0.00 | 0.00 | 10.90 |
| stdlib | 26 | 1.73 | 0.00 | 0.00 | 0.00 | 0.00 | 7.73 |
| string | 22 | 2.59 | 0.00 | 0.00 | 0.00 | 0.00 | 10.14 |
| time | 9 | 5.78 | 0.00 | 0.00 | 0.00 | 0.00 | 20.78 |
| Total | 98 | 2.41 | 0.11 | 0.07 | 0.00 | 0.00 | 10.80 |
wget and thttpd because, in these programs, several advanced library functions, e.g. wcwidth and strcasecmp_l, are called frequently. These functions are not included in our function-level instrumentation but can be instrumented on-demand without much effort. The glibc functions other than these 98 candidates are instrumented per instruction in the same way as user-code functions.

Then, we present the average number of summaries and taint rules for each candidate function classified by different headers in Table III. We categorize the summaries of each function into four types: input parameters to output parameters \((p_i \rightarrow p_o)\), input parameters to global variables \((p_i \rightarrow g_o)\), global variables to output parameters \((g_i \rightarrow p_o)\), and among global variables \((g_i \rightarrow g_o)\). In Table III, we take an approximation to count the number of variadic arguments as constant. For example, for printf(const char *format, ...), we count the number of arguments as two to derive the function summaries. We found most of the function summaries (236 out of 254) are from input parameter to output parameter, i.e., 2.41 on average of the 98 candidate library functions. The 11 summaries of type \(p_i \rightarrow g_o\) and 7 summaries of type \(g_i \rightarrow p_o\) are all from stdio.h. We have not found any dependency among global variables \((g_i \rightarrow g_o)\) caused by the candidate library functions. We generate 1058 taint rules from the function summaries. After deriving the summaries of the library functions, we manually check and confirm that the dependencies specified by the summaries are consistent with the official definition of the library functions [31]. Such static checks are coarse-grained since we do not address the runtime size of memory regions used as arguments in different calling contexts of the library function. More fine-grained function-level tainting effect evaluations are presented in Section V-C.

B. Efficiency of Sdft

We focus on evaluating the online phase of Sdft since the generation of the taint rules is modularized on individual functions and also offline without affecting the runtime overhead of DTA. With the instrumentation of the candidate C library functions, we investigate the efficiency of Sdft’s hybrid instrumentation. We compare the execution time of benchmark programs tracked by Sdft with the execution time tracked by Libdft64. The working task of each benchmark follows the standard SPEC2k6 workload or batch-job command options as mentioned at the beginning of Section V. Because the loading procedure does not introduce taint, we skip the instrumentation of ld-linux-x86-64.so for both tools.

The results are presented in Table IV. First, we profile the number of executed instructions of each benchmark program under the specific workloads, i.e., \#Total in Table IV. We run each benchmark program multiple times and record the average number of executed instructions. Libdft64 instruments all the instructions at runtime. In contrast, Sdft leaves a part of the executed instructions (in the 98 candidate library functions) uninstrumented. We profile the number of executed instructions inside the candidate library functions, i.e., \#Unins. in Table IV. These instructions are not instrumented at the instruction level. The number \#Unins. is different from the number of call-sites in Table II. Consequently, the proportions of these executed but uninstrumented instructions in the candidate library functions range between 5.3% and 36.4%. Specifically, taking 400.perlbench and Firefox for example, the low ratios of uninstrumented instructions (i.e., 8.6% and 5.3% respectively) indicate that the 98 candidates library functions are in relatively rare usage by these benchmarks, even though the call sites of these library functions can dominate the call sites of all the glibc functions in these benchmarks (i.e., 87.5% and 90.8% respectively in Table II).

Then, we record for each benchmark the original execution time (i.e. orig. in Table IV) and the execution times under the tracking of Libdft64 and Sdft respectively. We demonstrate the performance improvement of Sdft with different metrics. Specifically, Sdft achieves a 9.4%~69.9% time reduction and 1.10x~3.32x speedup on Libdft64. The average speedup is 1.58x. The slowdowns are reduced between 1.3x~24.6x. We find the ratios of uninstrumented instructions do not positively correlate with the performance improvement. For example, uninstrumenting 8.6% of the executed instructions for 400.perlbench can cause a 15.5% time reduction, while for MySQL, the time reduction is only 27.5% even we have uninstrumented 36.4% of the executed instructions. This is because different types of instructions take diverse time costs to propagate the taints. The instructions in the library functions used by MySQL are more likely in types that require less taint propagation time by Libdft64. The overall slowdowns reported in our work are not of the same magnitude as reported in [5] because we are working on 64-bit binaries and [5] reports slowdown on 32-bit benchmarks. The byte-level taint tracking on a 64-bit system introduces more complicated tainting operations on the shadow memory and shadow registers, which is time-consuming.

C. Library-side Tainting Effect of Sdft

Compared with the instruction-level taint tracking, function-level taint tracking of Sdft is more likely to introduce overapproximations in Tagmap. In this section, we investigate the tainting effect of different approaches on the candidate library functions. The objective is to confirm that the function-level instrumentation of Sdft has not triggered significant overapproximation compared to the instruction-level instrumentation of Libdft64.

We craft one small user program for each of the 98 library functions to launch one-time execution of the function using randomly generated inputs. We apply parameter-sized tainting on the inputs, which means for each parameter, we taint the entire parameter but not some specific byte of the parameter. We enumerate the choices of tainting strategy. Taking the summary of memcpy in Fig. 4 as an example, for the three input parameters para0–para2, we have \(2^3 = 8\) different choices to taint the shadow memory of one, two, or three of the parameters. Under each tainting strategy, the user program runs to call the library function and we track the data flow with Sdft and Libdft64 respectively. After the call-site of the library
function, we traverse the used pages of Tagmap, calculate the tainted bytes in Tagmap, average the number of tainted bytes of different strategies, and compare the results of Sdft and Libdft64. For space reason, we present the results of 20 library functions in Table IV and summarize the results of the 98 library functions in the last line. The average tainted space of Sdft’s function-level taint tracking is 1.02x of the space tainted by the instruction-level taint tracking of Libdft64.

Although function-level tainting tends to introduce overapproximation to tainted more space, We observe that in many of the 98 standard library functions, the size of space tainted by Sdft is however slightly smaller than the space tainted by Libdft64. This is because the instruction-level tainting will propagate the taints through local variables. Such local variables are intermediate volatile tainted locations on the stack or in some general-purpose registers, e.g., rcx, rdx. Such locations are not tainted by our function-level instrumentation. Besides, our PDG-based summaries include the control dependencies that are omitted by the instruction-level tainting of Libdft64 based on direct flows. Therefore in several library functions, e.g., strlen and puts, Sdft can track the implicit flows from the argument to the return value (in rax or xmm0) while Libdft64 ignores them. We count 26 of the candidate library functions whose return value is control-dependent to the tainted arguments as shown by the increase on the tainted return registers from 33 to 59 in Table IV. In these cases, tainting more bytes indicates mitigation of underapproximation of direct flow tracking, instead of introducing more over-approximation.

To address the correctness of the taint rules generated by the algorithms, we use a validation approach. The idea is that conducting the taint rules should produce a set of noninterference results. We check if the runtime tainting results are consistent with the expected noninterference property [32]. Specifically, we modify each user program to invoke a twin-executions of the candidate library function, i.e., to call the library function twice sequentially. In each pair of executions, we randomly choose the candidate input parameter to be tainted. From the taint rules of this function, we infer the tainted (high) and untainted (low) outputs. Then, we feed the two executions with the same untainted inputs but different tainted inputs.

We observe if the untainted outputs of the two executions differentiate to indicate a violation of noninterference. If the untainted outputs of the respective executions cannot be distinguished publicly under many random tainted inputs, we confirm the taint rules of the library function tend to comply with noninterference. For the example in Fig. 5, we call memcpy twice and choose the parameter src as tainted. In the context of student_cpy, we know the global stu.id is tainted but stu.score is untainted. Using the same n as the third parameter, we feed different random src and src’ as the second parameter of respective executions. By observing stu.score equals to stu’.score many times, we know the taint rules in Fig. 5 tend to enforce noninterference.
TABLE VI

| ID     | Program   | Type | #Instr_{Instr} (x 10^3) | #Instr_{Instr} (x 10^3) | T(s) |
|--------|------------|------|-------------------------|-------------------------|------|
| BID-6240 | wsm3p5     | HC   | 2.18                    | 1.38                    | 1.1  |
| CVE-2001-0414 | ntpd       | RCE  | 957.25                  | 436.61                  | 2.4  |
| CVE-2004-2093 | rsync     | RCE  | 5.37                    | 3.33                    | 0.3  |
| CVE-2005-1019 | Aeom      | HC   | 2.07                    | 1.42                    | 0.3  |
| CVE-2010-4221 | protfpd   | 5-OFF | 21.93                  | 13.12                   | 1.5  |
| CVE-2013-2028 | nginxx    | 5-OFF | 14.89                  | 10.97                   | 1.3  |
| CVE-2013-4788 | glibc     | 5-OFF | 1.80                    | 1.16                    | 0.4  |
| CVE-2016-9112 | openjpeg2 | 1-DIV | 672.07                | 653.62                  | 3.7  |

D. Effectiveness of Sdft’s Dynamic Data-Flow Tracking

We validate the effectiveness of Sdft by tracking the real-world vulnerabilities of CVEs triggered by public exploits. The CVEs are taken from related works [5], [8], [12], as presented in Table VII. We only report the results of the CVEs successfully deployed in our experimental environment, especially fitting the glibc version (2.27) and the LLVM required by the PDG. Out-of-date CVEs only deployable on old systems are not addressed. The validated types of vulnerabilities include remote code execution, stack overflow, division-by-zero, and heap corruption. For each case, we develop an individual Pintool over Sdft to track the vulnerability. We retrieve the inject point of the target program including the specific syscalls, functions, or program arguments as taint sources. We treat the vulnerable functions reported in the CVEs, including the suspicious variables and parameters in these functions as taint sinks. In several cases, the taint sources are not given straightforward and we have to investigate the exploit program to identify the potential taint sources. The results demonstrate that the taint tracking of Sdft can detect the sensitive flows corresponding to all the exploits of CVEs listed in Table VII. Meanwhile, we present in Table VI the number of instructions instrumented by Libdft64 and Sdft during the vulnerability tracking, as well as the time cost of Sdft. On CVE-2016-9112, we did not find a significant difference in the performance of Libdft64 and Sdft because openjpeg2 rarely calls standard C library functions. Instead, it calls frequently to libopenjp2.so and libm.so. It needs to be emphasized that we have not developed Pintools for the use-after-free vulnerabilities thus cannot deal with the CVEs reported in Table 2 of [12].

VI. DISCUSSION

In this section, we discuss several limitations and threats to the validity of our approach.

a) Function pointer abstraction. The PDG-based function summarization cannot effectively resolve the function pointer as the parameter or the global function pointer usage in the library function, because the static analysis on PDG cannot decide the control transfer targets of the indirect calls. Fine-grained binary-level CFG generations, e.g., TypeArmor [33] and BPA [34], can be used to refine the PDG and support the reachability analysis inside the callees. Currently, for the library functions that use indirect calls, e.g., bsearch, qsort and atexit declared in stdlib.h, we have to return to the instruction-level instrumentation that can check if the tainted data are used in the indirect calls at runtime by instrumenting at specific instructions, e.g., jmp, call, or ret.

b) Conservativeness of function-level abstraction. The static analysis of library functions cannot decide the runtime bound of memory regions passed as the argument, therefore the taint rules iteratively apply the propagation over each element of the memory region. This policy brings in over-approximation when only some element of an array parameter is tainted. We have evaluated that the overly approximate effect in the library functions is limited and it does not affect the taint tracking of vulnerabilities (Section V-C and V-D). To mitigate the over-approximation, global data-flow analysis is needed to bridge the contexts of the library function calls and the parameters to infer the range or element in the array being tainted.

c) Incompleteness of instruction support. Because the online phase of Sdft is an extension of Libdft64, both tools take the same instruction set support. For example, at the instruction level of user code, we ignored implicit flows and register EFLAGS. Therefore, even our function-level abstraction takes control-flow dependency as an option, the overall data-flow tracking is still limited to tracking explicit flows. However, we believe our approach can be further combined with the architectural-agnostic approach [8] which has much less limitation on the ISA supports.

d) Generability of the approach. Sdft requires the source code of functions to build the PDGs. This limitation forces us to focus on the library functions because their source codes are obtainable. The source code is used in the offline phase. The online phase of data-flow tracking does not need source code. Therefore the requirement has no impact on the deployment of DTA. Our instantiated instrumentation gets considerable improvement by instrumenting 98 of 1833 glibc-functions. Instrumenting more library functions reduces runtime overhead but raises longer loading time of library shared objects as stated in Section V-A. Consequently, Sdft works efficiently when complicated library functions are called intensively for a long time and the loading time becomes minor or ignorable.

VII. RELATED WORK

The system-wide dynamic taint analysis generally depends on virtual machine monitor [35], system/hardware emulator [36]–[40], or hardware-based techniques [41], or FPGA [17], [42], [43]. Dynamic binary instrumentation frameworks are widely used in tracking data flows in the address space at runtime. TaintCheck [1] uses Valgrind [44] and TaintTrace [2] uses DynamoRIO [45] to instrument binary for detecting overwrite attacks and format string attacks. LIFT [3] uses another DBI framework StarDBT [46] on Windows, with optimizations to track taints. Dyta[4] is a dynamic taint analyzer based on the Pin framework [25]. The framework...
relates taint to the EFLAGS register and propagating taints through implicit flows based on the CFG and post-domination information. DTA++ identifies the implicit flows that potentially cause under-tainting and generates targeted taint propagation rules using CFG to resolve the under-tainting. To make the DBI-based analysis more general, there are efforts to model the data flow among multiple processes or model the taint propagation for different target ISAs.

Performance overhead is the critical issue of DBI-based dynamic taint analysis. The improvements come from two aspects. The first is to offload the data tracking from program execution to introduce parallelization over different CPUs, hosts, processes, or threads. ShadowReplica decouples the taint analysis from the execution of binary with a shadow thread that communicates with the application thread using a lock-free ring buffer structure. TaintPipe and StraightTaint use symbolic formulas to model taint transfer on straight-lined operations for clearing the memory taint for memory stores. It requires no taint computation for registers and only a write to slow path when the loaded data are tainted. TaintRabbit proposes a generic taint analysis that supports more complex state-of-the-art DBI-based instruction-level taint analysis. Sdft uses reachability analysis on modular interprocedural PDGs to derive the library function summaries for the control- and data dependencies between the inputs and outputs of the function. Then Sdft generates the taint rules for each target library function. The taint rules are then used by a taint tracker of library functions. The library taint tracker is integrated into the DTA framework Libdft64, resulting in a more efficient dynamic taint analyzer. We apply Sdft on the standard C library functions of glibc to validate the efficiency and effectiveness of our approach. Future work includes resolving the function pointer and indirect calls used in the library functions, inferring precise memory bound of arguments with data-flow analysis, applying our approach to domain-specific third-party libraries, and developing more Pintools over our framework to mitigate more complicated vulnerabilities, e.g., UAFs.

APPENDIX A

DEFINITION OF PDG AND ALGORITHMS IN DETAIL

Algorithm 3 Source Nodes Derivation of Function p

| Procedure | SourceNodes(funStmtp, PDGp, Gp) |
|-----------|--------------------------------|
| Suppose | funStmtp \equiv function(t₁, ..., tₙ₋₁, tₙ) |
| candidates \leftarrow {} |
| for \( i = 0..n - 1 \) do | candidates.add(findNode("FORMAL_IN: i tᵢ', p), tᵢ)) |
| for all gStmt \in Gp do | |
| Suppose | gStmt \equiv extern τ gvar |
| candidates.add(findNode("GLOBAL_VALUE:@gvar', p), τ)) |
| \( Nᵢ' \leftarrow {} \) |
| for all \( (nᵢ, τᵢ) \in candidates \) do | |
| if isPrimType(τᵢ) then | |
| \( Nᵢ'' \leftarrow add(nᵢ) \) |
| \( p \leftarrow add(nᵢ \rightarrow nᵢ) \) |
| else if \( τᵢ \equiv cons(id, ...) \) then | |
| \( ins_k ← \%var = getelemptr inbounds * τᵢ, τᵢ, ... \) |
| \( nₓ ← findNode(insₓ, p) \) |
| for all \( τⱼ \in primTypeMap.get(id) \) do | |
| \( insⱼ ← load τⱼ, τⱼ, \%var, align sizeof(τⱼ) \) |
| \( nᵧ ← findNode(insᵧ, p) \) |
| if findPath(nᵢ, nₓ, nᵧ) \neq {} \land findNextUse(insₓ) = insᵧ then | |
| \( Nᵢ'' \leftarrow add(nᵢ) \) |
| \( p \leftarrow add(nᵧ \rightarrow nᵢ) \) |
| return \( Nᵢ'' \) |

The definition of PDG is mainly generalized from \[47\]. For each library function \( p \), the program dependency graph \( G_p \equiv (N_p, E_p) \) consists of the set of nodes \( N_p = N_{para} \cup N_{glb} \cup N_{call} \cup N_{ret} \cup N_{func} \cup N_{gfunc} \) and the set of edges \( E_p = E_{odep} \cup E_{ddep} \cup E_{para} \cup E_{cfunc} \). The types of nodes and edges are respectively represented in Table VII and Table VIII. For the parameter with struct type, the \( \text{FORMAL_IN}/\text{FORMAL_OUT}/\text{ACTUAL_IN}/\text{ACTUAL_OUT} \) nodes are further refined into the type node with the field nodes of struct. In such case, the edges of \( p$\_\text{fld}$ \) defines such refinement relation. Meanwhile, the edges of \( p$\_$\_\text{in}/p$\_\text{out} \) apply to the relation of type nodes of struct, and the edges of \( p$\_$\_\text{form}/p$\_\text{act} \) apply to the relation of field nodes of struct.

We develop two procedures, i.e., SourceNodes(funStmtp, PDGp, Gp) and TargetNodes(funStmtp, PDGp, Gp), to derive the source nodes \( N_p^s \), target nodes \( N_p^t \), and the mapping...
relation $\varphi$ in Section III-C. Algorithm 3 derives the source nodes in $N^s$ and the related $\varphi$ relation. In the algorithm, the set $\textit{candidates}$ of $(\text{node}, \text{type})$ pairs holds each potential node of parameters/global variables with its type information. Algorithm 4 identifies the target nodes $N^t$ representing the output of function $p$, as well as the related $\varphi$ relation. The list $\textit{candidate}$ holds the pointer-type global variables and parameters. $\textit{isPointer}(\tau)$ / $\textit{isPrimPointer}(\tau)$ decides if $\tau$ is a (primitive) pointer type. Algorithm 5 is the summary construction algorithm mentioned in Section III-C.

### REFERENCES

[1] J. Newsome and D. X. Song, “Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software,” in NDSS’05. The Internet Society, 2005.

[2] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige, “TaintTrace: Efficient flow tracing with dynamic binary rewriting,” in ISCC’06. IEEE Computer Society, 2006, pp. 749–754.

[3] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu, “LIFT: A low-overhead practical information flow tracking system for detecting security attacks,” in MICRO-39. IEEE Computer Society, 2006, pp. 135–148.

[4] J. A. Clause, W. Li, and A. Orso, “Dytan: a generic dynamic taint analysis framework,” in ISSTA’07. ACM, 2007, pp. 196–206.

[5] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, “libdft: practical dynamic data flow tracking for commodity systems,” in VEE’12. ACM, 2012, pp. 121–132.

[6] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, “DTA++: dynamic taint analysis with targeted control-flow propagation,” in NDSS’11. The Internet Society, 2011.

[7] H. C. Kim, A. D. Keromytis, M. Covington, and R. Sahita, “Capturing information flow with concatenated dynamic taint analysis,” in ARES’09. IEEE Computer Society, 2009, pp. 355–362.

[8] Z. L. Chua, Y. Wang, T. Baluta, P. Saxena, Z. Liang, and P. Su, “One engine to serve ‘em all: Inferring taint rules without architectural semantics,” in NDSS’19. The Internet Society, 2019.

[9] K. Jee, G. Portokalidis, V. P. Kemerlis, S. Ghosh, D. I. August, and A. D. Keromytis, “A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware,” in NDSS’12. The Internet Society, 2012.

[10] “libdft64,” 2019. [Online]. Available: https://github.com/AngoraFuzzer/libdft64

[11] S. Mallisery, Y. Wu, C. Hsieh, and C. Bau, “Identification of data propagation paths for efficient dynamic information flow tracking,” in SAC’20. ACM, 2020, pp. 92–99.

[12] J. Galea and D. Kroening, “The taint rabbit: Optimizing generic taint analysis with dynamic fast path generation,” in ASIA CCS’20. ACM, 2020, pp. 622–636.

[13] M. Stamatiougiannakis, P. Groth, and H. Bos, “Looking inside the blackbox: Capturing data provenance using dynamic instrumentation,” in 5th International Provenance and Annotation Workshop, IPAW’14, ser. Lecture Notes in Computer Science, vol. 8628. Springer, 2014, pp. 155–167.

[14] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giumfrida, and H. Bos, “Vuzzer: Application-aware evolutionary fuzzing,” in NDSS’17. The Internet Society, 2017.

[15] V. Jain, S. Rawat, C. Giumfrida, and H. Bos, “TIFF: using input type inference to improve fuzzing,” in ACSA’18. ACM, 2018, pp. 505–517.

[16] A. Quinn, D. Devecestr, P. M. Chen, and J. Finn, “JetStream: clusterscale parallelization of information flow queries,” in OSDI’16. USENIX Association, 2016, pp. 451–466.

[17] H. Kannan, M. Dalton, and C. Kozyrakis, “Decoupling dynamic infor-
Algorithm 4 Target Nodes Derivation of Function \( p \)

 procedure TARGETNODES\((\text{funStmt}_p, \text{PDG}_p, \mathcal{G}_p)\) 

 Suppose \( \text{funStmt}_p \equiv \text{function}(n_0, \ldots, n_{\tau_{ret}}, \tau_{ret}) \) \( p \) 

 \( N^p_0 \leftarrow \{ n_0 \} \) 

 \( \varphi.\text{add}(n_0 \rightarrow n_0) \) 

 candidates \( \leftarrow \emptyset \) 

 for \( i = 0 \ldots n - 1 \) do 

 if isPointer\((\tau_i)\) then 

 candidates.\text{add}(\text{findNode}(\text{GLOBAL_IN}: i, \tau_i, \tau_i))\) 

 for all \( glbStmt \in \mathcal{G}_p \) do 

 Suppose \( glbStmt \equiv \text{extern } \tau \text{ var} \) 

 \( n \leftarrow \text{findNode}(\text{GLOBAL_VALUE}: \text{var})\) 

 if isPrimitive\((\tau)\) then 

 \( n' \leftarrow \text{findNode}(\text{store } \tau, \tau \ast \text{gvar}, \text{align } \text{sizeof}(\tau), \tau)\) 

 \( N^p_0.\text{add}(n') \) 

 \( \varphi.\text{add}(n' \rightarrow n) \) 

 else 

 candidates.\text{add}(n, \tau)\) 

 for all \( (n_1, \tau) \in \text{candidates} \) do 

 if isPrimitive\((\tau)\) then 

 \( \text{ins}_s \leftarrow \text{getelempt in bounds} \tau, \tau, \ldots \) 

 \( n_x \leftarrow \text{findNode}(\text{ins}_s, \tau)\) 

 \( \text{ins}_s \leftarrow \text{store } \tau, \tau \ast \text{var}, \text{align } \text{sizeof}(\tau)\) 

 \( n_y \leftarrow \text{findNode}(\text{ins}_s, \tau)\) 

 \( n_x \leftarrow \text{findNode}(\text{ins}_s, \tau)\) 

 if (findPath\((n_s, n_x) \neq \emptyset \) \&\& \text{findNextUse}(\text{ins}_s) = \text{ins}_s) \text{then} 

 \( N^p.\text{add}(n_x, \tau) \) 

 \( \varphi.\text{add}(n_x \rightarrow n_s) \) 

 else if \( \tau \equiv \text{pointer}(\text{cons}(\text{id}_d, \ldots)) \) then 

 \( \text{ins}_s \leftarrow \text{getelempt in bounds} \tau, \tau, \ldots \) 

 \( n_x \leftarrow \text{findNode}(\text{ins}_s, \tau)\) 

 for all \( \tau_j \in \text{primTypeMap}.\text{get}(\text{id}_d) \) do 

 \( \text{ins}_s \leftarrow \text{store } \tau_j, \tau_j \ast \text{var}, \text{align } \text{sizeof}(\tau_j)\) 

 \( n_y \leftarrow \text{findNode}(\text{ins}_s, \tau_j)\) 

 if (findPath\((n_s, n_x) \neq \emptyset \) \&\& \text{findNextUse}(\text{ins}_s) = \text{ins}_s) \text{then} 

 \( N^p.\text{add}(n_y, \tau) \) 

 \( \varphi.\text{add}(n_y \rightarrow n_s) \) 

 return \( N^p_0, \varphi \)

Algorithm 5 Summary Generation of Function \( p \)

 procedure SUMMARYGEN\((N^p_0, N^p_{\tau_0}, \varphi)\) 

 summaries \( \leftarrow \emptyset \) 

 for all \( n_s \in N^p_0 \) do 

 for all \( \tau \in N^p_{\tau_0} \) do 

 if findPath\((n_s, n_x) \neq \emptyset \) \&\& \( \varphi.\text{add}(n_s, \tau) \) then 

 summaries.\text{add}(\varphi.\text{add}(n_s, \tau))\) 

 return summaries


---

Reformatted into a clean, readable text format.