podft: On Accelerating Dynamic Taint Analysis with Precise Path Optimization

Zhiyou Tian*, Cong Sun*, Dongrui Zeng†+, and Gang Tan‡
*State Key Lab of ISN, School of Cyber Engineering, Xidian University, China
Email: 21151213645@stu.xidian.edu.cn, suncong@xidian.edu.cn
†Palo Alto Networks, Inc., Santa Clara, CA, USA
Email: dzeng@paloaltonetworks.com
‡The Pennsylvania State University, University Park, PA, USA
Email: gtan@psu.edu

***Abstract*—Dynamic taint analysis (DTA) has been widely used in security applications, including exploit detection, data provenance, fuzzing improvement, and information flow control. Meanwhile, the usability of DTA is argued on its high runtime overhead, causing a slowdown of more than one magnitude on large binaries. Various approaches have used preliminary static analysis and introduced parallelization or higher-granularity abstractions to raise the scalability of DTA. In this paper, we present a dynamic taint analysis framework podft that defines and enforces different fast paths to improve the efficiency of DBI-based dynamic taint analysis. podft uses a value-set analysis (VSA) to differentiate the instructions that must not be tainted from those potentially tainted. Combining the VSA-based analysis results with proper library function abstractions, we develop taint tracking policies for fast and slow paths and implement the tracking policy enforcement as a Pin-based taint tracker. The experimental results show that podft is more efficient than the state-of-the-art fast path-based DTA approach and competitive with the static binary rewriting approach. podft has a high potential to integrate basic block-level deep neural networks to simplify fast path enforcement and raise tracking efficiency.**

I. Introduction

Dynamic taint analysis (DTA), also known as dynamic data-flow tracking (DDFT) or dynamic information-flow tracking (DIFT), is a program analysis technique that tracks selected data at runtime and checks specific tainted data for reaching sinks. Dynamic taint analysis can track system-wide or application-level taints. The *system-wide DTA* approaches are general but heavy-weighted tainting solutions that usually depend on virtual machine monitors [23], whole-system or hardware emulators [12], [20], [41], [49]–[51], or FPGA [19], [28], [52]. Compared with the complicated hardware extensions and the system-level monitors or emulators, *application-level DTA* is more flexible and pervasively used to investigate the vulnerabilities of an independent program. In this scenario, the tracking code can be instrumented during compilation [10], [14], [33], [36], [45] when source code is available, or instrumented on bytecode [21] or binaries [16], [18], [27], [29], [40], [42], [54].

Dynamic binary instrumentation (DBI) is promising in tracking the runtime taint propagation across a process's memory space. Such approaches hold the tainting status of suspicious data within tagging memory and virtual registers and check at particular program execution points to decide if specific runtime policies are enforced or violated. TaintCheck [40] uses Valgrind [39], and TaintTrace [16] uses DynamoRIO [13] to rewrite binaries for detecting various attacks. LIFT [42] instruments binary with StarDBT [11] on Windows. It first proposes to reduce the runtime overhead by the *fast paths* and *merged checks* to simplify the instrumented taint operations. libdft [5], [29] and its descendant reimplementation [8] are popular DBI-based taint tracking tools with a moderate slowdown to the native execution. On generalizing the DBI-based taint tracking, efforts are made to model the implicit flows [18], [27], the flows among distributed processes [31], or the taint propagations for different target ISAs [17].

Performance overhead is the prominent obstacle in using dynamic taint analysis. At the software level, the potential improvements come from two mostly orthogonal aspects. First, the parallelization-based overhead mitigation offloads the taint-tracking operations from the original program execution, introducing parallelization over different CPU cores [24], [37], [38], [44] or hosts [43]. Second, hybrid analyses may use static analysis to abstract the inlined taint propagation operations from per instruction to a higher granularity [10], [22], [25], [26], [42], [47], [53]. The function-level summaries [26], [53] can abstract the taint computations and reduce the overhead of instruction-level taint tracking. The library function abstractions [26] can be derived automatically with reachability analysis on the LLVM-PDG [7], [34] of the target library and integrated into libdft. Jee et al. [25] use per-basic-block analysis to express the tainting behavior with an algebra. The optimized operations are aggregated into maximized instrumentation units to minimize interference with the target program. *Fast path* [10], [22], [42], [47] generally instruments check-and-switch mechanisms at specific program locations to ensure a switch to an efficient sub-trace execution whose live variables are untainted. The original fast paths [42] are the code segments whose live-in and live-out states are determined safe at their entry point. Saxena et al. [47] analyze the untainted local variables and registers using a

+ Corresponding author

Workshop on Binary Analysis Research (BAR) 2023
3 March 2023, San Diego, CA, USA
ISBN 1-891562-84-3
https://dx.doi.org/10.14722/bar.2023.23010
www.ndss-symposium.org

modular stack analysis. Their fast path requires no tainting for registers and only a write to clear the memory taint for memory stores. Only memory loads are checked for a potential switch to the slow path when the tainted data are loaded. Iodine [10] profiles representative executions to develop an efficient predicated static taint analysis for the frequently-executed fast path, but it will introduce high recovery cost at invariant failures. Compared with the static fast paths, Taint Rabbit [22] generates the fast paths just-in-time even when the taints are presented at *in*- and *out*-states of basic blocks. The efficiency of their approach assumes the executions of a basic block are frequently over the same taint state.

Value-set analysis (VSA) [9] is a general-purpose static analysis that over-approximates the values for each variable at different program locations. Saxena et al. [47] first mentioned using value-set analysis to optimize dynamic data-flow tracking. However, they use a modular stack analysis on a simple abstract domain instead of the high-cost VSA on global and heap memory. TaintPipe [38] only uses VSA to resolve the memory access scope specified by the path predicate of straight-line code. SELECTIVETAINT [15] uses value-set analysis to ensure a must-not-tainted instruction set and taints only the instructions outside this set with static binary rewriting. Although SELECTIVETAINT is very effective in eliding uninvolved instructions, their static binary rewriting technique may bloat the attack surface of binaries and confront more scalability issues than DBI.

This paper presents podft, a dynamic taint analysis framework that combines the VSA-based must-not-tainted (MNT for short) instructions identification with a flexible fast-path-guided dynamic binary instrumentation approach for efficient taint tracking. The runtime fast paths of podft are either on the basic block level or the function level. These fast paths can be divided into three categories. The *function fast path* summarizes the tainting behavior of frequently-used library functions. The *naive fast path* is on the basic blocks holding only MNT instructions. The *complex fast path* is directed by a runtime tag memory checking at the entry of the hot and potentially-tainted basic blocks. In contrast to these fast paths, the rest basic blocks are decided on *slow paths* by podft.

For high efficiency, podft uses static analysis to build the tracking policies used by the fast paths. We first derive the function-level policy by modeling the taint computation of library functions with taint rules [26]. Then, for the user-code basic blocks, we use the value-set analysis [15] to divide the instructions into MNT- or potentially-tainted instructions. We derive the policy for naive fast paths on the basic blocks of only MNT instructions. Considering the potentially-tainted basic block as the one containing some potentially-tainted instructions, the policy of complex fast path is enforced on the *hot* potentially-tainted basic block when the tainted memory at the block's entry has no intersection with the static analyzed memory locations used by the basic block. Such memory locations are obtained by a static analysis similar to the *Merged Check* of [42]. The runtime of podft takes these policies to instrument only a limited number of potentially-tainted instructions to track taints. Even if a slow path instead of a complex fast path is taken for a potentially-tainted basic block, the instruction-level VSA results can still ignore tracking specific MNT instructions in that basic block for efficiency.

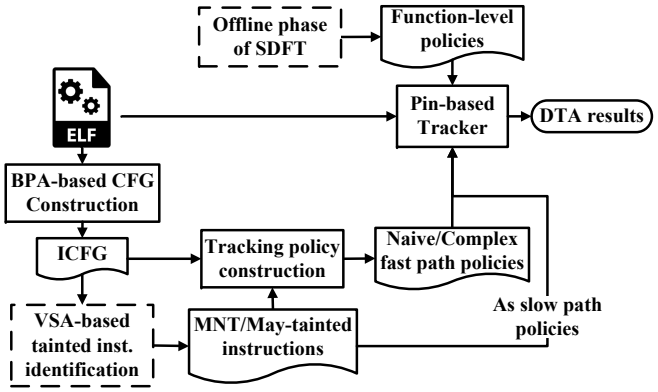

Fig. 1. Framework of podft (dashed block = usage of existing tools)

The contributions of this paper are summarized as follows:

1) We propose an efficient dynamic taint analysis framework podft. We design a novel approach using the instruction-level VSA results to build basic block-level tracking policies for our framework's naive and complex fast paths.
2) By enforcing the function- and basic block-level tracking policies with a Pin-based taint tracker and a prior static data-dependency analysis, our implementation outperforms the state-of-the-art DTA approach based on fast paths [22] and is competitive with the static binary rewriting approach [15].

II. DESIGN OF PODFT

This section depicts the framework of podft, including the interprocedural control-flow graph construction, the static analyses for the tracking policy construction and fast-slow path differentiation, and the dynamic Pin-based taint tracking.

A. System Overview

podft is a hybrid taint tracking framework, as presented in Fig. 1. Its static analysis uses the results of several existing analyses to build the tracking policies for the later runtime taint tracking. With the tracking policies, the Pin-based tracker takes fast paths on demand to track the taints' propagation at runtime efficiently.

First, podft uses BPA [32] to derive the indirect call targets for the binary-level interprocedural control-flow graph (ICFG). The ICFG is delivered to a value-set analysis [15] to derive the must-not-tainted instruction set $\mathcal{I}_u$. In this procedure, the instruction operands' value set is compared with the must-not-tainted (MNT) value set. If some of the operand's value is potentially tainted, the unknown value propagates, and the destination operand's value set is removed from the MNT value set. If all the operands' value sets are MNT value sets, the instruction is labeled as must-not-tainted instructions, and the MNT value set gets enlarged if the destination is concrete.

Then, podft's tracking policy construction module traverses the basic blocks of the ICFG to build the tracking policies for the runtime Pin-based taint tracker (Section II-B). We build the basic block-level fast path policy for the naive and complex fast paths and hold the instruction-level policy for the instructions on slow paths. Besides, we use the offline phase of [26] to generate the taint rules from the C standard library functions'

TABLE I. TRACKING POLICIES AND RELATED PREDICATES FOR INSTRUMENTATION

Policy Type	Policy applicable on
naive fast path	$\mathcal{B}_n = \{bbl \mid bbl \in \mathcal{B} - \mathcal{B}_l \wedge \forall ins \in bbl.ins \in \mathcal{I}_u\}$
complex fast path	$\{bbl \mid (bbl \in \mathcal{B}_c \cup \mathcal{B}_{lo}) \wedge (\mathit{TaintedMem}(\mathit{entry}(bbl)) \cap \mathit{MergedDep}(bbl) = \emptyset)\}$, s.t. $\mathcal{B}_c = \{bbl \mid bbl \in \mathcal{B} - \mathcal{B}_l \wedge (\mathcal{I}_t \cap bbl \neq \emptyset) \wedge \mathit{Hot}(bbl)\}$ and $\mathcal{B}_{lo} = \{bbl \mid bbl \in \mathcal{B}_l - \{libc.so\}\}$
slow path	$\mathcal{I}_s = \{bbl \mid bbl \in \mathcal{B} - \mathcal{B}_l \wedge (\mathcal{I}_t \cap bbl \neq \emptyset) \wedge \neg \mathit{Hot}(bbl)\} - \mathcal{I}_u$; or instructions in $\{bbl \mid (bbl \in \mathcal{B}_c \cup \mathcal{B}_{lo}) \wedge (\mathit{TaintedMem}(\mathit{entry}(bbl)) \cap \mathit{MergedDep}(bbl) \neq \emptyset)\} - \mathcal{I}_u$

```
void toy_test(int fd, char *buf, int size){
    int read_len = read(fd, buf, size); // taint source
    if(read_len > 0){
        printf("read data: %s\n", buf);
        for (int i = 0; i<2; i++){
            buf[i] = i;
            write(fd, buf[i], 1); // taint sink
        }
    }
}
int main(int argc, char *argv[]){
    char buffer[64] = {0};
    int fd = open(argv[1], O_RDWR);
    toy_test(fd, buffer, 64);
    return 0;
}
```

Fig. 2. Motivating Example

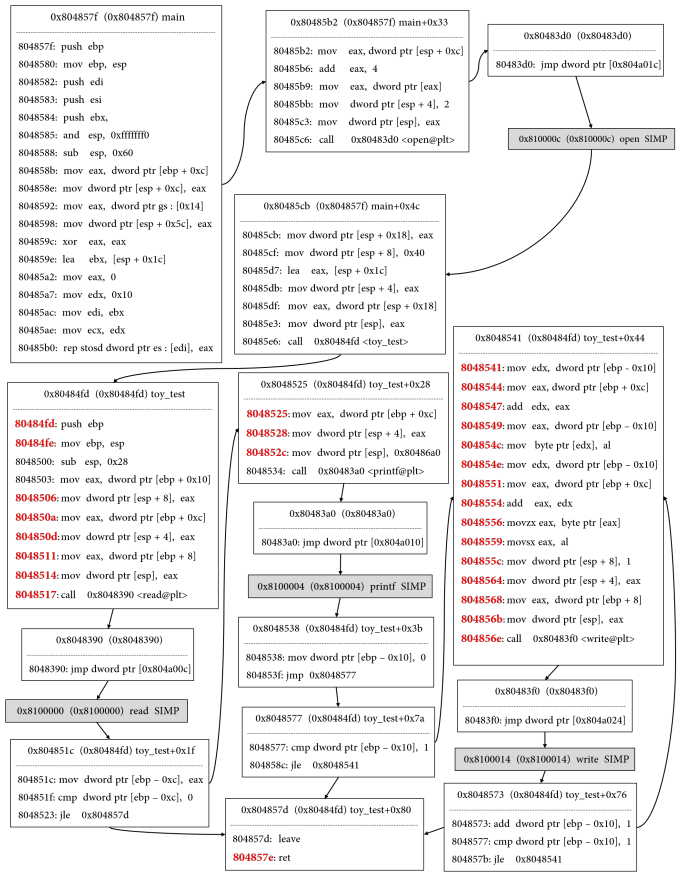

Fig. 3. Binary ICFG of the program in Fig. 2

program dependency graph (PDG). The taint rules serve as the function-level policies that abstract the taint computation of popular library functions.

Third, the Pin-based tracker of podft conducts the DBI-based dynamic taint analysis. We develop podft's dynamic analysis by extending the DTA tool libdft [5], [29], a Pintool developed on top of the Intel DBI framework Pin [35]. podft's tracker instruments the binary program with proper analysis routines based on the tracking policies (Section II-C). The function-level policy guides the tracker to flip to a function-level taint computation during the execution of library functions. The basic block-level policies are enabled on the traces of user code to take the naive fast path on MNT basic blocks and the complex fast path when the hot potentially-tainted basic blocks are decided untainted by a check on the tag memory state at the block entry. For the basic blocks decided tainted at runtime or potentially tainted but not frequently used, the addresses of taint-involved instructions are used to guide the tag operations along the slow paths.

B. VSA-based Tracking Policy Construction

The strategy of policy construction is based on the VSA-based instruction-level identification of the must-not-tainted and potentially-tainted instructions, i.e., $\mathcal{I}_u$ and $\mathcal{I}_t$ in [15], respectively. Let $\mathcal{B}$ be the basic blocks of the program's loaded code and $\mathcal{B}_l$ be the basic blocks of the library functions. Because our function-level tracking policy only applies to the C standard library, we distinguish the basic blocks of libc.so from the basic blocks of other libraries ($\mathcal{B}_{lo}$).

The basic block-level tracking policies for fast paths and instruction-level tracking policies for slow paths are presented in Table I. The tracking policies of *naive fast paths* are on basic blocks $\mathcal{B}_n$. If a basic block is identified as in $\mathcal{B}_n$, we ignore instrumenting tag operations on all its instructions. The tracking policies of *complex fast paths* are on $\mathcal{B}_c \cup \mathcal{B}_{lo}$. $\mathcal{B}_c$ stands for the user-code basic blocks *hot* in execution and have some potentially-tainted instructions. To generate $\mathcal{B}_c$, in our implementation, we determine a basic block as being hot if executed more than once and all its instructions are in $\mathcal{I}_t$. The library's basic blocks in $\mathcal{B}_{lo}$ fall in the complex fast path policy because the VSA-based analysis [15] cannot work on the libraries. We can only assume the instructions in these library basic blocks are potentially tainted. The applicability of complex fast-path policy is also decided by a runtime condition. Specifically, for a basic block bbl, we use static analysis to identify the entry-state memory and registers of bbl that are data-depended by the memory or registers of bbl, i.e., $\mathit{MergedDep}(bbl)$. Such entry-state memory and registers should not intersect with the runtime tainted memory and registers at bbl's entry state, i.e., $\mathit{TaintedMem}(\mathit{entry}(bbl))$. If this runtime condition is satisfied, we also ignore instrumenting the tag operations for this basic block. Otherwise, the basic blocks in $\mathcal{B}_c \cup \mathcal{B}_{lo}$ are enforced with the slow path tracking policy that instruments the instructions not in $\mathcal{I}_u$. Besides, the slow path policy applies to the potentially-tainted instructions in basic blocks of $(\mathcal{B} - \mathcal{B}_l \cup \mathcal{B}_n \cup \mathcal{B}_c)$, i.e., $\mathcal{I}_s$.

Input: *loaded_imanges*

1: **for all** *image*$\in$ *loaded_images* **do**
2: **if** *image* is libc.so **then**
3: *libc_scope* $\leftarrow$ *record_scope*(*image*);
4: **for all** $func \in$*image* **do**
5: *Instrument*$_{func}$(*entry*$(func)$, *exit*$(func)$);
6: **end for**
7: **else if** *image* is other library **then**
8: *libs_scope*$\leftarrow$*record_scope*(*image*);
9: **end if**
10: **end for**
11: $flag_{libc} \leftarrow 0$;
12: **for all** *trace*@runtime **do**
13: $flag_{libc} \leftarrow$ (*trace* $\in$ *libc_scope*)?1 : 0;
14: **for all** $bbl \in$*trace* **do**
15: **if** $flag_{libc} \vee bbl \in \mathcal{B}_n$ **then**
16: *skipTagOps*(*bbl*);
17: **else if** $bbl \in$*libs_scope* $\vee bbl \in \mathcal{B}_c$ **then**
18: **if** *TaintedMem*(*entry*(*bbl*))$\cap$ *MergedDep*(*bbl*)$=\emptyset$ **then**
19: *skipTagOps*(*bbl*);
20: **else**
21: **for all** $ins \in bbl - \mathcal{I}_u$ **do**
22: *Instrument*$_{\text{libdft}}(ins)$;
23: **end for**
24: **end if**
25: **else** $\{\forall ins \in \mathcal{I}_s\}$
26: **for all** $ins \in bbl - \mathcal{I}_u$ **do**
27: *Instrument*$_{\text{libdft}}(ins)$;
28: **end for**
29: **end if**
30: **end for**
31: **end for**

Fig. 4. Algorithm for Tracker's Dynamic Binary Instrumentation

Taking the program in Fig. 2 for example, `read` is a taint source that reads file `fd`'s input as taints to `buf`. The function `toy_test` prints `buf`, modifies the tainted `buf[0]` and `buf[1]` with constants in the for-loop, and delivers the untainted `buf[0..1]` to the taint sink `write` to modify the file. Thus, the data delivered to the sink are irrelevant to the tainted data from the source. This secure program helps illustrate the complex fast path decision. The binary ICFG of the program is presented in Fig. 3. The red code locations in Fig. 3 stand for the potentially-tainted instructions, and the rest are the MNT instructions identified by the VSA-based analysis. The PLT instructions (e.g., at 0x80483a0 for `printf`) are not tainted. The basic blocks with no potentially-tainted instruction, e.g., the one at 0x804857f, are in $\mathcal{B}_n$ for the naive fast paths. The basic block at 0x8048541 is executed twice and is in $\mathcal{B}_c$ to indicate a tentative complex fast path. Whether the basic block at 0x8048541 can take a complex fast path or a slow one is decided by the runtime checking on *TaintedMem*(0x8048541) $\cap$ *MergedDep*($bbl_{\text{0x8048541}}$). Besides, the potentially-tainted instructions in the basic blocks like 0x80484fd are in $\mathcal{I}_s$ to take the slow-path policy.

C. Pin-based Taint Tracking

The Pin-based tracker of podft is developed by extending libdft to skip instrumenting specific traces, basic blocks, or instructions according to the tracking policies in Table I. The DBI algorithm is presented in Fig. 4. For all the loaded images of the program, we record and differentiate the virtual address scopes of libc.so, other libraries, and user code. For the glibc functions, podft instruments tag memory operations at the function's entry and exit points based on the taint rules of [26] (*Instrument*$_{func}$ in Line 5). If the current trace is in libc.so or the current basic block is in $\mathcal{B}_n$, we skip instrumenting all instructions of this basic block. Otherwise, if the current basic block is in other libraries or $\mathcal{B}_c$, we decide if the basic block is on a complex fast path or a slow one. Suppose the registers and memory locations used in the basic block do not depend on the tainted memory and registers at the entry state of this basic block. In that case, we confirm the complex fast path and skip instrumenting all its instructions. Otherwise, we instrument all the potentially-tainted instructions with libdft's instruction-level instrumentation (*Instrument*$_{\text{libdft}}$ in Lines 22 and 27).

For the example in Fig. 3, the C standard library function `printf` is instrumented at the function level. The basic blocks of `printf` need no instrumentation at the instruction level. `open`, `read`, and `write` are system calls modeled by the I/O interface of libdft. The memory involved in the basic block 0x8048541, i.e., `buf[0]` and `buf[1]` pointed by `*[ebp+0xc]`, are sanitized with the constant of `i` held in `[ebp-0x10]`. Therefore *MergedDep*($bbl_{\text{0x8048541}}$) $=\emptyset$ at the entry of basic block 0x8048541. We have *MergedDep*($bbl_{\text{0x8048541}}$) $\cap$ *TaintedMem*(0x8048541) $=\emptyset$ at runtime when reaching 0x8048541. The memory locations used in this basic block have no data dependence with the tainted memory `buf` at the entry state of this basic block. Thus, the basic block 0x8048541 is decided to be a complex fast path, and all its instructions are not instrumented.

III. Implementation and Evaluation

This section presents several important issues in our implementation and evaluates podft's efficiency and effectiveness.

A. Implementation Issues

Finding the *hot* potentially-tainted basic blocks is critical to the decision of $\mathcal{B}_c$ and $\mathcal{I}_s$. In our implementation, a basic block is hot when executed more than once under the standard inputs, and all its instructions are potentially tainted. Therefore, we use another DBI execution over the standard inputs to identify the basic blocks executed more than once. After all, specifying a hot basic block is configurable and stand-alone prior to the DTA procedure. This instrumented execution also helps us to profile the related inputs and outputs of the hot basic blocks used in training the neural hot basic block model, as depicted in Section IV. podft holds a hashmap to facilitate efficient inclusion decisions in Fig. 4. The hashmap is generated using the results of VSA-based analysis identifying MNT- and potentially-tainted instructions. Each key of the hashmap is the address of a potentially-tainted basic block. Each value of the hashmap points to an instruction-address array of potentially-tainted instructions in the basic block. Queries on this structure can efficiently decide $\mathcal{B}_c, \mathcal{I}_s$, and $\mathcal{B}_n$ for tracking policy enforcement.

Another issue is deciding the complex fast path using static analysis to find the data dependencies between the memory/registers used by the basic block and the memory/register

TABLE II. BENCHMARK PROGRAMS

Dataset ID	Description
S1	10 SPEC CPU 2k6 benchmarks
S2	3 server programs, i.e., Nginx (1.22.0), Apache httpd (2.4.7), and MySQL (5.5.62)
S3	9 CVE programs, as presented in Table V

at the block's entry. By reimplementing the *Merged Check* of [42], we scan the traces of instructions and build the data dependencies for each memory reference. We use a fix-depth backward data-dependency graph traversal from the in-block memory references to find all the data dependencies to the basic block's entry memory locations. Then at runtime, a basic block entry check can decide if such entry memory locations have been tainted in the tag memory of podft.

B. Dataset and Experimental Settings

Our experiments are conducted on a Desktop with a 2.8GHz×4 Intel Core(TM) i7-7700HQ CPU, 8GB RAM, and Linux 3.16.0 kernel (Ubuntu 14.04 32-bit). The DBI framework is Pin v2.14, and libdft we used is [5]. Because the VSA-based MNT instructions identification [15] can only work on 32-bit binaries, we adapt the function-level taint rule generation [26], developed on libdft64 [8] for 64-bit binaries, to the 32-bit libdft [5]. The main adaption work is to reimplement the tag operations used by each taint rule.

The benchmarks we use to evaluate podft are presented in Table II. We compile the binaries of these benchmarks on their default compiler optimization level. For the performance evaluation, we use the standard working task in the DBI execution of each benchmark program of dataset S1 and S2. Specifically, for the SPEC2k6 benchmarks of dataset S1, we use the standard SPEC2k6 workload `test` in the instrumented execution. For httpd and Nginx in dataset S2, we use the benchmarking tool ab [1] to randomly select and deliver 10,000 and 100,000 requests to the servers. For MySQL, we use the load emulation tool mysqlslap [2] to connect the MyISAM [4] and InnoDB [3] storage engines of MySQL, automatically generate a complicated table, and issue 1,000 related SQL queries with 10 parallel clients. The dataset S3 is mostly collected from the related work [15]. We ensure that the vulnerabilities of these CVEs are reproduced in our experimental environment.

C. Efficiency of podft

We first compare podft's efficiency with other dynamic taint analysis approaches, including Taint Rabbit [22], Dytan [6], [18], Triton [46], and Taintgrind [30]. Taint Rabbit is developed on DynamoRIO [13], and Taintgrind is on Valgrind [39]. Dytan and Triton use Pin as used by podft. For deploying the related tools, we deploy Taint Rabbit with the taint policy TaintRabbit-ID (TR-ID) and TaintRabbit-BV (TR-BV). For a fair comparison, we set the same taint sources and sinks for each tool.

The results are presented in Table III. Besides the execution time of the instrumented benchmark programs tracked by different tools, the *slowdown* metric of each tool (SD in Table III) measures the magnification of each tool's instrumentation cost compared with the original execution. For the dataset S1 and S2, podft achieves slowdowns of 1.6x to 27.9x with an average slowdown of 10.6x. In contrast, the state-of-the-art fast-path-based tool, i.e., Taint Rabbit, has a 30x average slowdown for its TR-ID policy configuration and a 120x average slowdown for the TR-BV policy configuration. We can see that on each benchmark in Table III, podft outperforms other DTA tools. According to the average slowdowns, podft is more efficient than the other DTA tools. More specifically, TaintRabbit-BV reaches a timeout when tracking *429.mcf*. Dytan cannot finish the instrumented execution of *458.sjeng* properly. Triton is relatively inefficient, reaching timeouts on dataset S1 and four benchmarks of S2.

Then, because podft uses the VSA-based static analysis of SELECTIVETAINT [15], we compare the efficiency of podft with this static binary rewriting approach. To deploy the tools provided by SELECTIVETAINT, we deploy both the STATICTAINTALL and the SELECTIVETAINT system. STATICTAINTALL system instruments all the instructions with static taint analysis. First, we confront difficulties using SELECTIVETAINT's static rewriting on the server binaries of dataset S2. Therefore we only use the benchmarks in S1 for this evaluation. On dataset S1, podft has an average slowdown of 12.5x, while STATICTAINTALL and SELECTIVETAINT report average slowdowns of 39.1x and 27.1x, respectively. SELECTIVETAINT reduces the slowdowns of STATICTAINTALL in all the S1 benchmarks. Although podft is generally more efficient than SELECTIVETAINT on S1, we observe that in several cases, e.g., *400.perlbench* and *429.mcf*, SELECTIVETAINT can outperform podft.

We also compare the basic block-level instrumentation coverage of podft's DBI and SELECTIVETAINT's static binary rewriting on dataset S1 and the standard workload. We observed that podft has an average instrumentation coverage of 40.38%, while SELECTIVETAINT's average instrumentation coverage is 53.73%. SELECTIVETAINT statically instruments many basic blocks that are unreachable by the benchmark executions. We cannot compare the runtime reachable basic blocks of SELECTIVETAINT-rewritten binaries with the podft-instrumented basic blocks because the static rewriting has introduced additional code structures into the binary.

D. Effectiveness of podft's Dynamic Taint Analysis

We validate the effectiveness of podft by tracking the real-world vulnerabilities of CVEs triggered by public exploits. The CVEs and the tracking time of podft are presented in Table V. podft's taint tracking can successfully detect the sensitive-flow-related exploits of all these CVEs. For each CVE, we develop a specific Pintool over podft to track the vulnerability. To retrieve the taint sources, we identify the target binary's specific syscalls, functions, or program arguments. We also treat the user input as taint sources in the cases where the taint source is unspecified by the CVE. We identify the taint sink by manually analyzing the assembly code and investigating the vulnerable functions reported by the CVEs. We analyze the suspicious variables and the instructions that operate on these variables in these functions. Consequently, the instrumented check decides if such a variable holds a taint tag at these instructions' locations. We observed that the tracking procedure of podft tainted all the vulnerable sinks.

TABLE III. PERFORMANCE IMPROVEMENT OF PODFT COMPARED WITH OTHER DTA APPROACHES ON DATASET S1 AND S2. SLOWDOWN $SD_{TOOL}=T_{TOOL}/T_{ORIG}$, S.T. TOOL=PODFT, TAINTRABBIT-ID, TAINTRABBIT-BV, TAINTGRIND, DYTAN, TRITON

Benchmark	T_{orig} (s)	podft		TaintRabbit-ID		TaintRabbit-BV		Taintgrind		Dytan		Triton	
		T_{podft}(s)	SD_{podft}	T_{ID}(s)	SD_{ID}	T_{BV}(s)	SD_{BV}	$T_{Taintgrind}$(s)	$SD_{Taintgrind}$	T_{Dytan}(s)	SD_{Dytan}	T_{Triton}(s)	SD_{Triton}
400.perlbench	4.3	120.1	27.9	126.6	29.4	204.9	47.7	346.7	80.6	3,208.6	746.2	Timeout	N/A
401.bzip2	5.4	46.4	8.6	85.7	15.9	91.3	16.9	1,634.9	302.8	20,812.6	3,854.2	Timeout	N/A
429.mcf	2.5	5.2	2.1	34.2	13.7	Timeout	N/A	193.6	77.4	2,845.4	1,138.2	Timeout	N/A
445.gobmk	15.3	128.2	8.4	219.2	14.3	204.7	13.4	6,039.4	394.7	45,147.5	2,950.8	Timeout	N/A
456.hmmer	2.2	55.6	25.3	74.8	34.0	78.5	35.7	998.6	453.9	11,531.6	5,241.6	Timeout	N/A
458.sjeng	4.6	28.3	6.2	34.9	7.6	39.2	8.5	1,355.8	294.7	Failed	N/A	Timeout	N/A
462.libquantum	0.5	3.6	7.2	6.8	13.6	8.2	16.4	22.9	45.8	333.2	666.4	Timeout	N/A
464.h264ref	12.6	191.8	15.2	762.3	60.5	2,588.7	205.5	5,057.7	401.4	81,798.6	6,492.0	Timeout	N/A
471.omnetpp	0.4	9.1	22.8	54.6	136.5	54.1	135.3	259.6	649.0	1,401.5	3,503.8	Timeout	N/A
473.astar	9.1	14.5	1.6	129.6	14.2	129.2	14.2	1,467.4	161.3	19,525.1	2,145.6	Timeout	N/A
mysqlslap_myisam	0.6	4.5	7.5	27.5	45.8	404.2	673.7	77.5	129.2	951.9	1,586.5	Timeout	N/A
mysqlslap_innodb	0.7	4.9	7.0	28.8	41.1	392.3	560.4	73.8	105.4	999.1	1,427.3	Timeout	N/A
httpd_req_10k	0.6	5.3	8.8	11.7	19.5	15.9	26.5	26.7	44.5	227.6	379.3	40,509.2	67,515.3
httpd_req_100k	4.9	27.4	5.6	39.8	8.1	74.4	15.2	219.3	44.8	2,386.9	487.1	Timeout	N/A
Nginx_req_10k	0.5	5.5	11.0	9.8	19.6	11.7	23.4	23.9	47.8	201.1	402.2	33,671.6	67,343.2
Nginx_req_100k	4.2	20.1	4.8	24.7	5.9	32.8	7.8	196.4	46.8	2,062.3	491.0	Timeout	N/A
Avg. slowdown		–	10.6	–	30.0	–	120.0	–	205.0	–	2,100.8	–	67,429.3

TABLE IV. PERFORMANCE COMPARISON OF PODFT WITH SELECTIVETAINT

Benchmark	podft	STATICTAINTALL		SELECTIVETAINT	
	SD_{podft}	T_{All}(s)	SD_{All}	T_{Select}(s)	SD_{Select}
400.perlbench	27.9	26.8	6.2	24.2	5.6
401.bzip2	8.6	263.1	48.7	174.1	32.2
429.mcf	2.1	39.1	15.6	3.8	1.5
445.gobmk	8.4	543.4	35.5	538.5	35.2
456.hmmer	25.3	161.4	73.4	129.2	58.7
458.sjeng	6.2	126.8	27.6	94.8	20.6
462.libquantum	7.2	6.3	12.6	4.5	9.0
464.h264ref	15.2	1,255.9	99.7	801.2	63.6
471.omnetpp	22.8	16.4	41.0	14.9	37.3
473.astar	1.6	277.4	30.5	61.7	6.8
Avg. slowdown	12.5	–	39.1	–	27.1

TABLE V. EFFECT OF PODFT ON TRACKING VULNERABILITIES OF CVES (S-OF=STACK OVERFLOW, H-OF=HEAP OVERFLOW)

ID	Program	Type	T_{podft}(s)
CVE-2021-41253	Zydis v3.2.0	H-OF	1.4
CVE-2019-8354	SoX v14.4.2	H-OF	2.8
CVE-2018-19655	dcraw v9.28	S-OF	1.7
CVE-2018-11575	ngiflib v0.4	S-OF	1.4
CVE-2018-6612	jhead v3.00	H-OF	1.2
CVE-2017-1000437	Gravity v0.3.5	S-OF	9.8
CVE-2017-14411	MP3Gain v1.5.2	S-OF	2.2
CVE-2013-2028	Nginx v1.4.0	S-OF	1.1

IV. WORK-IN-PROGRESS

podft is a dynamic taint analysis framework that enforces multiple fast-path tracking policies for efficient taint tracking. podft uses a VSA-based analysis and function-level tainting abstraction to generate these tracking policies. We use a basic block-level static data-dependency analysis to decide whether a hot potentially-tainted basic block takes a complex fast path or a slow path. Such analyses benefit the taint tracking efficiency but still have several limitations:

1) The VSA-based analysis [15] cannot work on library code. The function-level summaries [26] work only for popular library functions. Thus, we must over-approximate most library basic blocks as all-instruction-tainted slow paths to enforce instruction-level slow-path tracking policy.
2) In several cases, the data-dependency analysis may need to be more precise in identifying the block-used memory references. In such cases, the hot basic blocks must be overly approximated as slow paths.

The above limitations motivate us to develop a more scalable complex fast path policy enforcement approach. Our solution is to use deep neural networks to track taints at the basic-block level. Since we can identify the hot potentially-tainted basic blocks, these basic blocks are more frequently executed and have more inputs and outputs. We can use the fuzzing technique to profile the runtime tag memory relations between the inputs and outputs. Because these hot basic blocks only use a limited number of memory and registers, such relations can be properly abstracted and are not expected over complicated.

Taking the principle of Neutaint [48], we are trying to train neural hot basic block embeddings using these tag memory relations. Unlike Neutaint, which uses only dynamic program embedding, our neural hot basic block embedding will also consider several instruction-level static features to merge the taint-tracking prediction of different hot basic blocks into one deep model for each binary program. Integrating our design into podft can avoid differentiating complex fast paths from slow paths on hot basic blocks, thus avoiding the static data-dependency analysis. Moreover, compared with Neutaint's program-level embeddings, our basic block-level embeddings are more flexible to be used in traditional DTA infrastructures. We also plan to develop more dynamic neural library-function embeddings to optimize the tainting abstraction of [26] and the function-level policy enforcement of podft. Another subsistent issue we need to resolve is that several static code features are used in our neural basic block embeddings, making our deep model correlate more with the compiler options of the binary than Nuetaint's model.

ACKNOWLEDGMENT

Zhiyou Tian and Cong Sun were supported by the National Natural Science Foundation of China (62272366) and the Key Research and Development Program of Shaanxi (2023-YBGY-371).

REFERENCES

[1] "ab - Apache HTTP server benchmarking tool." [Online]. Available: https://httpd.apache.org/docs/2.4/programs/ab.html

[2] "mysqlslap — A Load Emulation Client." [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/mysqlslap.html

[3] "The InnoDB Storage Engine." [Online]. Available: https://dev.mysql.com/doc/refman/5.7/en/innodb-storage-engine.html

[4] "The MyISAM Storage Engine." [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/myisam-storage-engine.html

[5] "libdft," 2012-2022. [Online]. Available: https://gitlab.com/brown-ssl/libdft/

[6] "Dytan Taint Analysis Framework on Linux 64-bit," 2014. [Online]. Available: https://github.com/behzad-a/Dytan

[7] "Program Dependence Graph in LLVM," 2017. [Online]. Available: https://bitbucket.org/psu_soslab/program-dependence-graph-in-llvm

[8] "libdft64," 2019. [Online]. Available: https://github.com/AngoraFuzzer/libdft64

[9] G. Balakrishnan and T. W. Reps, "Analyzing memory accesses in x86 executables," in *CC'04*, ser. LNCS, vol. 2985. Springer, 2004, pp. 5–23.

[10] S. Banerjee, D. Devecsery, P. M. Chen, and S. Narayanasamy, "Iodine: Fast dynamic taint tracking using rollback-free optimistic hybrid analysis," in *SP'19*. IEEE, 2019, pp. 490–504.

[11] E. Borin, C. Wang, Y. Wu, and G. Araujo, "Software-based transparent and comprehensive control-flow error detection," in *CGO'06*. IEEE, 2006, pp. 333–345.

[12] E. Bosman, A. Slowinska, and H. Bos, "Minemu: The world's fastest taint tracker," in *RAID'11*, ser. LNCS, vol. 6961. Springer, 2011, pp. 1–20.

[13] D. Bruening, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, Massachusetts Institute of Technology, 2004.

[14] W. Chang, B. Streiff, and C. Lin, "Efficient and extensible security enforcement using dynamic data flow analysis," in *CCS'08*. ACM, 2008, pp. 39–50.

[15] S. Chen, Z. Lin, and Y. Zhang, "SelectiveTaint: Efficient data flow tracking with static binary rewriting," in *30th USENIX Security Symposium*. USENIX Association, 2021, pp. 1665–1682.

[16] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige, "TaintTrace: Efficient flow tracing with dynamic binary rewriting," in *ISCC'06*. IEEE, 2006, pp. 749–754.

[17] 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.

[18] J. A. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *ISSTA'07*. ACM, 2007, pp. 196–206.

[19] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: a flexible information flow architecture for software security," in *ISCA'07*. ACM, 2007, pp. 482–493.

[20] A. Ermolinskiy, S. Katti, S. Shenker, L. Fowler, and M. McCauley, "Towards practical taint tracking," UCB/EECS-2010-92, UC Berkeley, Tech. Rep., 2010.

[21] X. Fu and H. Cai, "FlowDist: Multi-staged refinement-based dynamic information flow analysis for distributed software systems," in *30th USENIX Security Symposium*. USENIX Association, 2021, pp. 2093–2110.

[22] 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.

[23] A. Ho, M. A. Fetterman, C. Clark, A. Warfield, and S. Hand, "Practical taint-based protection using demand emulation," in *EuroSys'06*. ACM, 2006, pp. 29–41.

[24] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis, "ShadowReplica: efficient parallelization of dynamic data flow tracking," in *CCS'13*. ACM, 2013, pp. 235–246.

[25] 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.

[26] X. Kan, C. Sun, S. Liu, Y. Huang, G. Tan, S. Ma, and Y. Zhang, "Sdft: A PDG-based summarization for efficient dynamic data flow tracking," in *QRS'21*. IEEE, 2021, pp. 702–713.

[27] 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.

[28] H. Kannan, M. Dalton, and C. Kozyrakis, "Decoupling dynamic information flow tracking with a dedicated coprocessor," in *DSN'09*. IEEE, 2009, pp. 105–114.

[29] 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.

[30] W. M. Khoo, "Taintgrind: a Valgrind taint analysis tool," 2012. [Online]. Available: https://github.com/wmkhoo/taintgrind

[31] H. C. Kim, A. D. Keromytis, M. Covington, and R. Sahita, "Capturing information flow with concatenated dynamic taint analysis," in *ARES'09*. IEEE, 2009, pp. 355–362.

[32] S. H. Kim, C. Sun, D. Zeng, and G. Tan, "Refining indirect call targets at the binary level," in *NDSS'21*. The Internet Society, 2021.

[33] L. Lam and T. Chiueh, "A general dynamic information flow tracking framework for security applications," in *ACSAC'06*. IEEE, 2006, pp. 463–472.

[34] S. Liu, G. Tan, and T. Jaeger, "PtrSplit: Supporting general pointers in automatic program partitioning," in *CCS'17*, 2017, pp. 2359–2371.

[35] C. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI'05*. ACM, 2005, pp. 190–200.

[36] S. Mallissery, 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.

[37] J. Ming, D. Wu, J. Wang, G. Xiao, and P. Liu, "StraightTaint: decoupled offline symbolic taint analysis," in *ASE'16*. ACM, 2016, pp. 308–319.

[38] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu, "TaintPipe: pipelined symbolic taint analysis," in *24th USENIX Security Symposium*. USENIX Association, 2015, pp. 65–80.

[39] N. Nethercote and J. Seward, "Valgrind: A program supervision framework," *Electron. Notes Theor. Comput. Sci.*, vol. 89, no. 2, pp. 44–66, 2003.

[40] 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.

[41] G. Portokalidis, A. Slowinska, and H. Bos, "Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation," in *EuroSys'06*. ACM, 2006, pp. 15–27.

[42] 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, 2006, pp. 135–148.

[43] A. Quinn, D. Devecsery, P. M. Chen, and J. Flinn, "JetStream: cluster-scale parallelization of information flow queries," in *OSDI'16*. USENIX Association, 2016, pp. 451–466.

[44] O. Ruwase, P. B. Gibbons, T. C. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. P. Ryan, "Parallelizing dynamic information flow tracking," in *SPAA'08*. ACM, 2008, pp. 35–45.

[45] G. Ryan, A. Shah, D. She, K. Bhat, and S. Jana, "Fine grained dataflow tracking with proximal gradients," in *30th USENIX Security Symposium*. USENIX Association, 2021, pp. 1611–1628.

[46] F. Saudel and J. Salwan, "Triton: A dynamic symbolic execution framework," in *Symposium sur la sécurité des technologies de l'information et des communications*, ser. SSTIC, Rennes, France, Jun. 2015, pp. 31–54.

[47] P. Saxena, R. Sekar, and V. Puranik, "Efficient fine-grained binary instrumentationwith applications to taint-tracking," in *CGO'08*. ACM, 2008, pp. 74–83.

[48] D. She, Y. Chen, A. Shah, B. Ray, and S. Jana, "Neutaint: Efficient dynamic taint analysis with neural networks," in *SP'20*. IEEE, 2020, pp. 1527–1543.

[49] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *ASPLOS'04*. ACM, 2004, pp. 85–96.

[50] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August, "RIFLE: an architectural framework for user-centric information-flow security," in *MICRO-37*. IEEE, 2004, pp. 243–254.

[51] H. Yin, D. X. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *CCS'07*. ACM, 2007, pp. 116–127.

[52] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis, "Hardware enforcement of application security policies using tagged memory," in *OSDI'08*. USENIX Association, 2008, pp. 225–240.

[53] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "TaintEraser: protecting sensitive data leaks using application-level taint tracking," *ACM SIGOPS Oper. Syst. Rev.*, vol. 45, no. 1, pp. 142–154, 2011.

[54] E. Zhu, F. Liu, Z. Wang, A. Liang, Y. Zhang, X. Li, and X. Li, "Dytaint: The implementation of a novel lightweight 3-state dynamic taint analysis framework for x86 binary programs," *Comput. Secur.*, vol. 52, pp. 51–69, 2015.