FLOWMATRIX: GPU-Assisted Information-Flow Analysis through Matrix-Based Representation

Kaihang Ji[†], Jun Zeng[†], Yuancheng Jiang[†], Zhenkai Liang[†],
Zheng Leong Chua[‡], Prateek Saxena[†], Abhik Roychoudhury[†]

[†]*National University of Singapore* [‡]*Independent Researcher*

Abstract

Dynamic Information Flow Tracking (DIFT) forms the foundation of a wide range of security and privacy solutions. The main challenges faced by DIFT techniques are performance and scalability. Due to the large number of states in a program, the number of data flows in a program is prohibitively complex to reason about. Though data flow analysis at a larger scale is a common need, such as interactive queries about data flows in program execution, existing DIFT solutions typically focus on a subset of data flows to support practical applications. Data flow queries are challenging to perform efficiently using existing approaches. In this paper, we identifies the linear property in offline DIFT investigation under dependency-based information flow rules. The linear relationship enables a novel matrix-based representation, FLOWMATRIX, to represent DIFT operations concisely and makes it practical to adopt GPUs as co-processors for DIFT analysis. FLOWMATRIX provides efficient support for interactive DIFT query operations. We design a DIFT query system and prototype it on commodity GPUs. The extensive evaluation shows the dramatic performance improvement by outperforming CPU-based baseline by 5.6 times and rapid response of DIFT query in 2 seconds. It also has two to three orders of magnitude higher throughputs compared to typical DIFT analysis solutions. We also demonstrate the efficiency and efficacy of new DIFT query operations in the evaluation.

1 Introduction

Understanding information flows in program execution is the basis for advanced system diagnosis and attack response. Dynamic Information Flow Tracking (DIFT) is a commonly used solution that tracks data flows in programs, which identifies the flow of data among program states. It has a wide range of applications, such as vulnerability detection and attack investigation [1–19], data leakage detection [20–24], protocol recovery [25], and system configuration diagnosis [26, 27].

In a real-world program with a large number of states, the data flows in the program are prohibitively complex to be reasoned about. As a result, DIFT analysis is typically applied on a subset of data flows: selecting a few program states and tracking how they affect other program states along with the data flows in the program, e.g., taint analysis [1]. During DIFT analysis, investigators often need to run a large amount of DIFT checks. For example, analysts commonly encounter programs that prevent information flows from being tracked correctly [28], such as in an implicit information flow where variables are affected by other variables indirectly. When reasoning about such information flows, there is a need to probe information flows with multiple queries. As another example, in misconfiguration diagnosis, investigators need to check multiple information flows in a server program taking different configuration files to identify the root cause and ramifications of misconfiguration [26, 27]. Such query-style DIFT analysis is even more challenging, which requires heavy computing support [29].

There have been many efforts to improve the performance and scalability of DIFT techniques. From a high-level view, information flow analysis systems include two main tasks, *defining information flow rules for individual operations* and *applying the rules on a set of operations*. An active research direction of improving DIFT efficiency focuses on reducing and parallelizing the application of rules on the operation set. One category of work decouples flow tracking from program execution and offload the tracking in parallel to one or more CPU cores [30–36] or other host systems [29, 37, 38]. Another category of work improves DIFT performance by limiting flow tracking under certain conditions. For example, several solutions [3, 39, 40] perform fastpath which enables applications to execute without any instrumentation or taint propagation for those execution paths without involving any tainted metadata. Optimizing DIFT operations can also speed up analysis when DIFT tools summarize the semantics of a chunk of programs [21, 41] or eliminate redundant tracking logic code in hot paths [42].

Another research direction to improve DIFT operations focuses on optimizing the information flow tracking rules. In fact, the fundamental limitation of DIFT is caused by such

rules. Specifically, most of the existing solutions in taint analysis follow information flows defined by program semantics, where the complexity of the rules makes it difficult to scale up. Recent research has developed a new representation of DIFT operations for several different purposes, including speeding up taint analysis [41] or automatically generating rules for information flow tracking [43,44]. We aim to explore how these new representations can open up a new space for efficient DIFT analysis.

Our Insight. The data flow captures how variables in the input space (source) of a program affect program variables in the output space (sink). Recent researches show that they can be further represented as a dependency or gradient relationship [43,44]. Furthermore, we observe that the dependency relation is simpler in semantics when describing information-flow operations. Specifically, it forms a linear relation in a program trace, as we will analyze in Section 3. The linear relationship can be represented as *matrix* operations. As a well-studied mathematical structure, matrix operations are optimized in software libraries as well as in hardware implementations, such as GPUs, for performance improvements. The matrix representation and hardware support will optimize the interactive DIFT query applications in an after-the-fact style on a program trace.

Our Approach. We design FLOWMATRIX, a novel matrix-based representation of information flow operations that enables efficient and versatile DIFT analysis. DIFT operations using FLOWMATRIX can be efficiently processed by GPUs, which are specialized hardware designed to perform large-scale linear operations in parallel. The efficiency in data flow operation representation and the speed-up by GPU can efficiently support *interactive DIFT queries*. We also design a query system using FLOWMATRIX to support repeated DIFT queries.

We evaluate FLOWMATRIX on the aspects of efficiency and effectiveness. Using GPU as the co-processor, we showed that it can achieve 5.6 times speed-up compared with CPU DIFT baseline. We also demonstrate the efficiency in supporting DIFT query operations. It performed DIFT queries in less than 0.1s for common cases and in less than 2s for heavier real-world application. Moreover, we compare our tool with the state-of-the-art and commonly used DIFT analysis and query tools, namely, LibDFT [45] and JetStream [29]. Our approach outperforms LibDFT in three orders of magnitude in data flow tracking throughput, and achieves comparable throughtput to JetStream without using a server cluster. Finally, we adopt case studies to illustrate how our DIFT query can help to improve trace-based after-the-fact analysis, including handling implicit control flows and diagnosing server misconfiguration.

In summary, we made the following contributions in this work.

- We analyze offline dynamic information flow operations on binaries and recognize its linear property. We propose FLOWMATRIX, a novel way of representing DIFT operations using matrices, enabling off-the-shelves GPU to be used as a hardware co-processor for DIFT operations.
- We design an efficient solution to support interactive DIFT queries on offline execution traces. It also supports fine-tuning implicit flows and indirect flows efficiently.
- We prototype FLOWMATRIX and interactive query on commodity GPUs. We demonstrate the efficiency of DIFT operations and queries, where the DIFT query's response is in milliseconds.

2 Background

In this section, we briefly introduce existing DIFT techniques and analyze their limitations.

2.1 Dynamic Information Flow Analysis

Dynamic information flow tracking (DIFT) identifies the flow of data between two program locations. In taint analysis, it is often referred to as *(taint) sources* and *(taint) sinks*. For example, taint sources can be external inputs (e.g., network sockets, file reads, and user inputs), while taint sinks can be program outputs, sensitive memory area, and the program counter (PC), etc.

In order to track the flow of data in binary programs, the semantics of instructions need to be followed. Taking taint analysis as an example, for each instruction executed by the program, the metadata of the source operand is propagated to the destination operand based on the instruction semantics. The propagation step is then repeated for all instructions executed. At any point in time, the current state of the data can be obtained by observing the recorded metadata. In DIFT queries, the data flow of instructions is also needed for deciding the existence of queried flow. Tracking the data flow forms the main bulk of work performed by DIFT analysis. It is determined by a set of rules, which are generated manually [45–47] by domain experts or automatically by inference [43].

2.2 Limitations of Existing DIFT Rules

The DIFT rule representation directly decides the complexity of DIFT operations. We analyze the resulting limitations of existing DIFT rules.

The data-flow rules of state-of-the-art DIFT mechanisms are typically implemented in high-level programming languages. An detailed example of such implementation of different DIFT tools is provided in Figure 10 in Appendix. Due to the richness of semantics of the language describing the taint rules, it is challenging to further summarize the complex propagation logic, which are often *Turing complete*. While works like Taint Flow Algebra [41] simplify the expression of taint rules as algebraic expressions, the summary of such

$$al[0] \leftarrow al[0],\ al[1] \leftarrow al[1],\ ...,\ al[7] \leftarrow al[7]$$
$$al[0] \leftarrow\ bl[0],\ al[1] \leftarrow bl[1],\ ...,\ al[7] \leftarrow\ bl[7]$$
$$bl[0]\ \leftarrow\ bl[0],\ bl[1] \leftarrow bl[1],\ ...,\ bl[7] \leftarrow\ bl[7]$$

Figure 1: Dependency-based taint propagation rule for `or al, bl`.

expressions relies on term rewriting, which still requires non-trivial effort and is an open research problem.

The standard approach employed by current DIFT engines starts with an initial (taint) state and repeatedly applies a taint propagation function to it, accumulating in the final taint state specified by the analyst. Note that the result of any intermediate taint state is dependent on the previous state, and this dependency between the current and previous result forces the entire process to be sequential. The sequential [35] nature of the propagation process essentially prevents it from being parallelized.

Under such sequential constraints, it is difficult to propagate two instructions in parallel, because in each propagation, the output taint state has a dependency on the current taint state, which is the output of the last propagation.

3 FLOWMATRIX Approach

In this section, we analyze the core properties of DIFT operations and introduce a matrix-based representation. We show how it enables efficient DIFT operations and GPU-based processing in information flow analysis.

3.1 Matrix Representation of Information Flow Operations

We formally describe the terms related to data-flow tracking on dynamic execution traces. Let $\tau : \langle I_1, \cdots, I_N \rangle$ be the execution trace of a program, which is an ordered sequence of instructions, from I_1 to I_N. Let $S_i : \{0,1\}^n$ be a column vector, which is the taint state of the instruction I_i, represented as a bit vector of size n. Let $F_i : \{0,1\}^n \rightarrow \{0,1\}^n$ denote the DIFT rule for instruction I_i, modeled as a function that maps S_i to S_{i+1}. Let S_α denote the taint source and S_ω denote the taint sink where $1 \leq \alpha < \omega \leq N+1$. It follows that $S_\omega : S_{i+1} = F_i(S_i) \mid i \in \alpha..(\omega-1)$. DIFT operations analyze whether there are data flows between S_α and S_ω.

Our Insight. In the instruction `or al, bl`, it *ORs* the lowest eight bits in `ebx` register with the lowest eight bits in `eax` register and stores the result in the the lowest eight bits in `eax`. Its taint rule can be written in a dependency-based representation [43], shown in Figure 1. The left side of the arrow denotes the states in the destination of the instruction's operation, i.e., *uses*, while the right side denotes the states in the source of the instruction, i.e., *defs*. For simplicity, we have omitted the influence to the status register in the illustration.

In this instruction, the first input bit of *al* and *bl* influences the first output bit of *al*. The corresponding DIFT propagation rule can be written as:

$$al_{out}[1] := al_{in}[1] \vee bl_{in}[1].$$

which is actually a simplification of the more verbose form in boolean algebra:

$$\begin{aligned} al_{out}[1] = (1 * al_{in}[1]) + \cdots + (0 * al_{in}[8]) \\ +(1 * bl_{in}[1]) + \cdots + (0 * bl_{in}[8]). \end{aligned}$$

In the verbose form, we explicitly include all input and output taint bits of the instruction. The coefficient for each of the input bits determines if the particular bit has an effect on the output taint bit. We can represent the taint state of the instruction as a bit vector, $S : \{0,1\}^n$. Let $w_{j,i} : \{0,1\}$ represent the influence of $S_{in}[i]$ over $S_{out}[j]$, the rule can be generalized as:

$$S_{i+1}[j] = w_{j,1} * S_i[1] + w_{j,2} * S_i[2] + \cdots + w_{j,n} * S_i[n].$$

Therefore, the propagation function $F : \{0,1\}^n \rightarrow \{0,1\}^n$, is a system of n such equations:

$$\begin{aligned} S_{i+1}[1] &= w_{1,1} * S_i[1] + w_{1,2} * S_i[2] + \cdots + w_{1,n} * S_i[n], \\ S_{i+1}[2] &= w_{2,1} * S_i[1] + w_{2,2} * S_i[2] + \cdots + w_{2,n} * S_i[n], \\ &\vdots \\ S_{i+1}[n] &= w_{n,1} * S_i[1] + w_{n,2}, * S_i[2] + \cdots + w_{n,n} * S_i[n]. \end{aligned}$$

As a result, the taint rule of an instruction I_i can be represented as an $n \times n$ matrix:

$$M_i = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \cdots & w_{nn} \end{bmatrix}.$$

We call this representation FLOWMATRIX. Using the FLOWMATRIX representation, the propagation function F is defined as:

$$F_i : M_i \times S_i$$

With the definition of the propagation function F, we represent information flow propagation from source to sink as a sequence of function applications.

$$S_\omega = F_{\omega-1}(\cdots F_{\alpha+2}(F_{\alpha+1}(F_\alpha(S_\alpha)))).$$

As matrix multiplication is a composition of linear systems, the FLOWMATRIX representation enables an efficient summary of information flows. If M_a, M_b are two FLOWMATRIX

data flow matrices, their summarized data flow matrix M_{ab} can be calculated by multiplying two matrices:

$$M_{ab} = M_b \times M_a.$$

Thus, for the instructions sequence $< I_\alpha, \cdots, I_{\omega-1} >$, we can summarize their data flows as:

$$M_{summary} = M_{\omega-1} \times \cdots \times M_{\alpha+2} \times M_{\alpha+1} \times M_\alpha.$$

This enables the pre-computation of the propagation rule for instructions beforehand with $M_{summary}$.

$$\begin{aligned} S_\omega &= (M_{\omega-1} \times \cdots (M_{\alpha+2} \times (M_{\alpha+1} \times (M_\alpha \times S_\alpha)))) \\ &= (M_{\omega-1} \times \cdots M_{\alpha+2} \times M_{\alpha+1} \times M_\alpha) \times S_\alpha \\ &= M_{summary} \times S_\alpha. \end{aligned}$$

3.2 GPU-assisted FLOWMATRIX Operations

Matrix is a well-studied structure in parallel processing. As DIFT operations can be represented as matrix operations, it opens up an opportunity to scale up DIFT operations using hardware and algorithms developed for efficient matrix operations. For example, though originally designed for graphics processing, GPU is increasingly used as a modified form of stream processor (or a vector processor). The thousands of cores make the GPU a very promising choice for highly parallel applications like matrix and vector calculations.

To process the DIFT operation matrix in GPU, we need to represent the rule matrix R in a format accepted by GPU. Most modern languages (including C/C++ and CUDA) store 2D arrays/matrices as 1D arrays in a row-major layout. However, such a matrix layout incurs a heavy space overhead and has a cubic time complexity in matrix-matrix multiplication tasks. Given that a program state may contain millions of bits, both shortcomings hinder the direct application of this dense layout in DIFT with such an enormous matrix size.

To address this challenge, we make use of an observation that most rule matrices are extremely sparse – one bit has data flows towards only a few bits. For example, the matrix for an x86-64 instruction `mov rax, [rbp-8]` has 128 bitwise data flows: 64 flows from memory `[rbp-8]` to register `rax` and 64 succeeding flows from memory `[rbp-8]` to itself. As the matrix size is 128×128, its sparsity would be 99.2%. In light of the high sparsity, we store FLOWMATRIX as sparse matrices [48], where only those non-zero elements are stored, by their indices and values. Thus, the space complexity for sparse matrix storage is linear to a matrix's number of non-zeros (NNZ). The performance of sparse general matrix-matrix multiplication (SpGEMM) is much better than dense matrix-matrix multiplication with a high sparsity [49]. In addition, the matrices need to be normalized to fit the context of each other before multiplication. We provide further details in Appendix A.2.

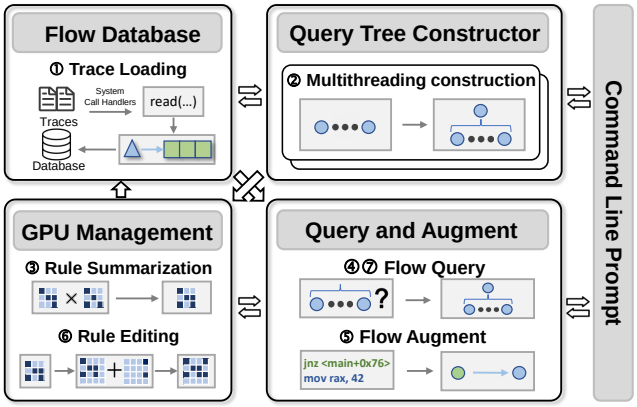


Figure 2: FLOWMATRIX Query architecture overview.

4 FLOWMATRIX-Based DIFT Query

In this section, we present our solution to enable offline efficient DIFT queries with FLOWMATRIX.

4.1 Overview

Figure 2 shows the design of our FLOWMATRIX-based DIFT query system. Given a program execution trace, it provides users with an interactive command line interface to query information flows repeatedly from various sources to destinations. Sources and destinations each are either (1) a collection of positions in the trace, or (2) a collection of system calls where our tool hooks their data flows automatically, or (3) a collection of instructions that involve specified registers or memory contents as their operands.

The FLOWMATRIX-based DIFT query system first loads the trace to a database for scalable processing (Step 1). It then prepares for queries by building a query tree (Step 2). The query tree constructor invokes back-end GPU management component for FLOWMATRIX summarization, where the instructions in the database are converted into FLOWMATRIX data flow matrices (Step 3). Once summarization is done, the query tree is ready for incoming DIFT queries in Step 4 and replies with information flow dependencies. Users are able to obtain arbitrary direct data flow dependencies.

If some of the dependencies are not explicitly tracked by direct data flows, which is a challenging scenario for traditional DIFT analysis [28], FLOWMATRIX DIFT query supports efficient information flow augment for such two common cases: indirect data flows and implicit control flows. We show that the data flow represented by FLOWMATRIX can be augmented (Step 5), which is further discussed in Section 4.3. Our tool internally calls GPU management unit to perform FLOWMATRIX editing on GPUs (Step 6). Finally, the augmented flows are ready for user query (Step 7).

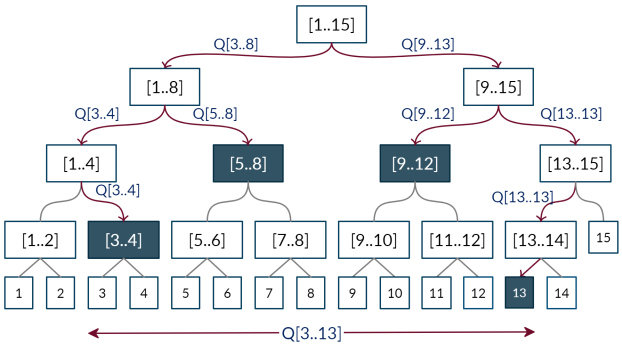


Figure 3: A query tree of 15 leaf vertices handles a query from 3 to 13. The tree splits the query into sub-queries to search for pre-computed data flows of segments. Those recursive sub-queries are presented as red edges marked with sub-query ranges. Dark vertices are chosen ones in this query to be multiplied.

4.2 Query Tree Construction

The problem of finding data flow using FLOWMATRIX can be formulated as follows: Given a sequence of data flow matrices M_1 ... M_N, and two positions α and ω, find data flows between position α and position ω as matrix products $M_\alpha \cdot M_{\alpha+1} \cdot \ldots \cdot M_\omega$. After DIFT operations are abstracted as matrix operations, optimization can be done through leveraging the mathematical structure of matrix. Matrix operations are associative, i.e., $(A \times B) \times C = A \times (B \times C)$, which allows the out-of-order computation of matrix multiplication. As a result, instead of performing a sequence of N matrix multiplications, we can compute the matrix product in a pairwise manner, which requires $log(N)$ operations. This enables us to parallelize the process of matrix multiplication, and thus increase overall throughput.

We use a binary-tree data structure to coordinate the range data flow queries over a trace, borrowing the idea of a segment tree. A query tree is a full binary tree, where each vertex other than leaves has exactly two child vertices. Each vertex stores a data flow matrix of a sub-segment. For instance, the root vertex stores the data flow from the beginning to the end of the whole segment $[1..N]$, and its children represent two halves $[1..N/2]$ and $[N/2+1..N]$ respectively. We keep splitting all segments into two halves until all of them reach size 1. In Figure 3, we show an example of such a binary-tree built based on a snippet with 15 instructions. Every non-leaf vertex has been marked with a segment it represents: the summary of two child vertices.

Tree construction is the pre-computation process before queries. We take the execution trace as input and compute the data flow stored in every on-tree vertex. As each non-leaf vertex relies on the data flows from both children, we adopt a recursion algorithm to build the tree from bottom

Algorithm 1: FLOWMATRIX Tree Construction

```
// trace - An execution trace of a program
// α, ω - Indices in trace, where α ≤ ω
// v - A vertex in the binary query tree
// root - Root vertex in the binary query tree
// Return with summarized matrices replied from
   children.
1  function BuildSubTree(trace, v, α, ω)
2      v.segment ← [α...ω];
3      if α = ω then
           // v is a leaf vertex
4          v.M ← ruleCache.loadMatByIdx(trace, α);
5      else
           // v is a non-leaf vertex
6          v.lChild ← ϕ;
7          v.rChild ← ϕ;
8          BuildSubTree(trace, v.lChild, α, v.mid);
9          BuildSubTree(trace, v.rChild, v.mid + 1, ω);
10         v.M ← matrixBuilder.Summarize(v.lChild.M, v.rChild.M);
11     end
12     return v
13 end
14 function QueryTreeConstruct(trace)
15     root ← ϕ;
16     root ← BuildSubTree(trace, root, 1, trace.length());
17     return root
18 end
19 function GetSubFlow(v, α, ω)
20     if v.segment ⊂ [α...ω] then
           // In range. Return this vertex's matrix.
21         return v.M
22     end
23     if v.segment ∩ [α...ω] = ϕ then
           // Fully outside. Return an identity matrix.
24         return I
25     end
       // Overlapping! Ask children recursively.
26     leftSum ← GetSubFlow(v.lChild, α, ω);
27     rightSum ← GetSubFlow(v.rChild, α, ω);
       // Return with summarized matrices replied from
          children.
28     return matrixBuilder.Summarize(leftSum, rightSum)
29 end
30 function GetIntervalFlow(root, α, ω)
       // Get data flow matrix for segment [α..ω].
31     return GetSubFlow(root, α, ω)
32 end
```

to top. More specifically, as function `QueryTreeConstruct` shown in Algorithm 1, we start assigning tasks from the full segment, $[1..N]$. Every time we divide the current segment into two halves (if it has not yet become a segment of length 1), and then call the same procedure on both halves. When a segment's size reaches 1, then it is a leaf and represents a data flow of a single instruction. The data flow matrix of one instruction is directly loaded from the data flow cache or database. Then, the parent vertex summarizes data flow matrices from two children, and uses the product matrix as the data flow matrix stored in the parent. Similarly, data flow matrices of other non-leaf vertices are computed after their children are both ready. Once the root is ready, construction of the whole query tree construction is completed.

Figure 3 also illustrates an example where the data flow between interval $[3..13]$ is queried from a trace of length 15. Our solution procedure starts from the root $[1..15]$: The query interval is a subset of the segment of the root. Thus, we move to child vertices and split the interval into $[3..8]$ and $[9..13]$

logically. We repeat these steps until we meet a vertex whose segment is a subset of the query interval, e.g., [3..4]. Then the data flow matrix for [3..4] will be returned. Finally, to obtain the result of the query, we just summarize those returned matrices, which are $M_{3\to4}$, $M_{5\to8}$, $M_{9\to12}$ and $M_{13\to13}$ in our example.

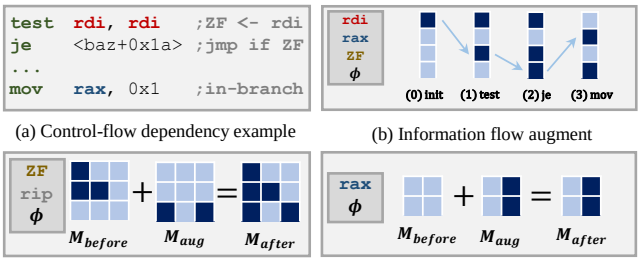
Note that at any level, one query will have at most two overlaps intervals, the most left interval and the most right interval. Only those overlaps would cause calls to the next level on the tree. Otherwise, the function call will directly return with the vertex's data flow matrix or an identity matrix. In other words, *Query* function at most visits four vertices on one level; among those four, at least two of them will directly return without diving into their subtrees. Given that a query tree is a balanced binary tree and that its height is $\lceil log_2 N \rceil$ following $O(logN)$, the number of total vertices we have to visit in a query is $O(logN)$. Note that only when there is an overlapping, the data flow summarization will be performed. Thus, a query would only operate data flow summarization for $O(logN)$ times.

4.3 FLOWMATRIX Extension for Implicit Flows and Indirect Flows

Tracking of implicit and indirect data flows, of which two common examples are memory address references and conditional control flows [28], has always been a challenge in DIFT. These non-explicit data flows are usually defined in a taint propagation policy [50]. The challenge arises due to the severe over-tainting (taint explosion) that resulted from propagating all non-explicit data flows. Typical solutions use propagation policies based on heuristics, such as one-level table lookups [51] to control the conditions of implicit taint propagation, or specifically handled strict control flow dependence [52]. For existing DIFT solutions, the problem is exacerbated by the fact that taint propagation policies have to be determined before DIFT is performed and any changes to the policies necessitates a rerun of the DIFT making iterative fine tuning of the policy hard. This highlights two desirable properties we would like to achieve: a unified way of reasoning and defining non-explicit data flow and a cheap way to perform changes to taint propagation policies.

Implicit and indirect data flows typically imply an indirect influence. Specifically, implicit flows happen when the condition variable in a conditional branch implicitly affects the data that is defined in the branch; indirect flows exist because a memory address indirectly affects the data at the memory location. Intuitively, we can use a variable ϕ to make these implicit/indirect relation explicit. The taint propagation policy then can be defined as a set of ϕ where enabling the tracking of an implicit data flow can be achieved by incorporating ϕ in the taint propagation.

FLOWMATRIX enables this to be done in a unified and low-cost manner. Assigning an additional variable is an 1-row-



(a) Control-flow dependency example

(b) Information flow augment

(c) Augment for *je* instruction

(d) Augment for *mov* instruction

Figure 4: Example for implicit flow augment. V_{snap} is the extended virtual variable for data flow snapshot. All matrices have been simplified for easy presentation.

column extension operation on FLOWMATRIX: the column vector represents the source of influence for ϕ and the row vector denotes what is being influenced. After modification of one or several data flow matrices, our query tree design allows minimum vertex updates. Only ancestors of modified vertices are no longer carrying the up-to-date data flows. Thus, a query tree update is to re-compute a chain of ancestors of a modified vertex.

Specifically, implicit flow and indirect flow are handled as follows: An implicit flow is represented by defining the dependency from the condition variable to ϕ and from ϕ to any variables being defined by in-branch instructions; An indirect flow is represented by defining the dependency from a memory address to ϕ and ϕ to the memory content.

Examples in X86 We provide an X86 example of the implicit control flow in Figure 4(a). In this case, there is an implicit control flow between register *rdi* and register *rax*: *rdi* contains data from external inputs and controls a conditional jump which leads to a data move instruction, assigning register *rax* with a constant. The condition variable here is the Zero Flag (*ZF*) set by the `test` instruction; the control flow influences *rax* implicitly. As a result, we assign an implicit flow variable, and set data flows from *ZF* to the variable and from the variable to *rax*.

Another X86 example is provided for the memory address reference in Figure 5(a). There is an implicit flow between register *rcx* and the target of an indirect call through a memory reference. The memory address variable in the move instruction is equivalent to *rsi*; the *def* variable here is register *rdx*. Since all of these are performed in a single instruction, we can optimize it by directly adding data flows from *rsi* to *rdx* explicitly in this move instruction.

Figure 4(c) and Figure 4(d) illustrates the matrix edition operations in FLOWMATRIX for the control-dependency case shown in Figure 4(a). As for those cases where no variable is assigned explicitly, matrices are directly modified by adding the originally-missed influences to them. Figure 5(b) illustrates the matrix operation for such optimized scenarios.

```
lea  rsi,QWORD PTR[rdi+rcx]
mov  rdx,QWORD PTR[rsi]
call rdx
```

(a) Example in memory reference

(b) Augment for *mov* instruction

Figure 5: Example for indirect flow augment. All matrices have been simplified for easy presentation.

5 Implementation

We have prototyped FLOWMATRIX utilities and query framework[1]. To provide a friendly interface, we also provide users the ability to raise influence queries and view results through an *interactive* command line prompt. In this section, we introduce important technical details during our implementation.

Choice of GPU Programming Framework. FLOWMATRIX needs to be represented in sparse matrix format. There are many libraries available for GPU-based sparse matrix supports, including TensorFlow [53], PyTorch [54], and CuSparse [55]. TensorFlow is an end-to-end open-source platform for machine learning but also provides APIs for direct tensor operations. However, it currently does not support SpGEMM, which is crucial for FLOWMATRIX. PyTorch is also a platform mainly designed for machine learning. With extension package [56], PyTorch is able to support SpGEMM operations. However, due to the high-performance overhead from Python, and lack of flexibility to customize GPU management, we decide to provide PyTorch-based Python bindings for query only. For better performance, we implement our prototype based on CuSparse library from the NVIDIA CUDA toolkit. This provides us an opportunity to pursue better performance but requires us to manage GPUs by ourselves.

Multi-threading and Database Management. Due to GPUs' powerful ability in parallel computing, a single thread usually can not fully utilize their computing power in the task of query tree construction. Fortunately, we observe that different sub-trees of a FLOWMATRIX query tree do not depend on each other. Thus, it enables multiple threads to work on several sub-trees in parallel. Specifically, the ancestor vertices of sub-trees that are not yet computed can be later summarized by one process within negligible time after all workers finish their jobs.

After a thorough analysis of the performance profile of a worker, we discover that most of the time had been spent on database loading and storing and data transferring between CPU and GPU. We notice that in a sub-tree construction, a worker always loads data flow rules from the database sequentially as it performs a DFS-like algorithm. Inspired by this, our tool deploys a read cache which reads a sequence of instruction data flows from the database at one time. Also, a write buffer is adopted for inserting summarized data flows into the database within one transaction.

GPU Scheduling. Without scheduling, even summarizing two FLOWMATRIX data flows is performed into fully dependent stages, where each stage never starts until its previous stage finishes. For example, we first load the matrix from the database, transfer it to GPU memory, and then do the same thing to the second one. Such a pipeline is unnecessary as these two matrices do not have dependencies. In this way, GPU resources are also not well-scheduled among threads, as all threads may be transferring data at the same time while GPU cores are idle.

To address this issue, we use CUDA streams, which allow programs to overlap computation and data transfer operations insides or across threads to maximize the utilization of GPU cores. In our previous case, two matrices can be transferred to GPU memory asynchronously. As for multiple threads, the GPU cores work in a pipeline manner: once a computation task of one thread is done, cores automatically move on to the next task from another thread.

Data Flow Rules. Data flow rules taken by FLOWMATRIX are automatically generated by TaintInduce [43], which infers taint rules from observations of CPU states. Not all generated rules can be directly transformed into a matrix representation. In particular, for those branch instructions and conditional move instructions, TaintInduce provides different taint rules corresponding to different branches. When FLOWMATRIX summarizes data flows into matrices, it requires a unique rule for a particular instruction. Thus, in execution traces, besides instruction sequences, we also record necessary branch condition information stored in registers (e.g., `eflags` register) and pick the corresponding rule during the data-flow summarization.

Extensible APIs. We aim to expose most of the tool's options through a command line prompt, and we also support power users who may need more flexibility than what provided commands can offer. Our tool provides direct access to its functionality and allows users to customize their own commands. To use these utilities, users would write command handler functions in C++ and register them to the command prompt. Inside the functions, users can invoke our tool's API to acquire specified information flows, inspect its content, modify it and write back to the database at will.

6 Evaluation

In this section, we present the evaluation of the following aspects of FLOWMATRIX.

[1] Source code is available at https://github.com/mimicji/FlowMatrix.

- The performance of FLOWMATRIX-based DIFT query operations, as well as the improvement achieved by GPU assistance;
- The throughput of the DIFT query enabled by FLOWMATRIX works on real-world programs compared with traditional dynamic taint analysis;
- The performance bottleneck and space overhead of FLOWMATRIX-based DIFT query;
- The benefit of FLOWMATRIX demonstrated via case studies.

6.1 Experiment Setup

We conduct all experiments on a server with an Intel E5-2620 v4 CPU processor, two NVIDIA Tesla V100 GPUs and 256GB physical memory. The OS is Ubuntu 16.04 LTS. The CPU processor is equipped with eight 2.10GHZ cores, 16 hyper threads, and 20MB L3 cache.

We use 15 common programs of which vulnerabilities are listed in Table 1 as our security task dataset. Our program selection follows criteria: (1) covering common categories of vulnerability, namely *Stack Overflow*, *Heap Overflow*, *Out-of-bound Read & Write*, *Divide by Zero*, *Information Leakage*, and *Null Pointer Dereference*; (2) selecting vulnerable programs with publicly available proof-of-concept exploits. Their vulnerabilities are triggered by exploits we found from links on NVD [57] and recorded with a simple DynamoRIO [58] instruction tracer we implemented. Our modular design allows users to replace this tracer with any other one as long as it provides instruction traces, memory access addresses, `eflags` register at data flow branches (e.g., `cmov` instructions) and system call traces. A C++ API with a pre-defined structure has been provided for users to convert other traces to our format.

Moreover, we also prepare an additional experiment dataset of seven real-world applications to generally evaluate FLOWMATRIX DIFT query performance:

- **tar** - archive a text file and untar,
- **gzip** - compress a text file and decompress,
- **bzip** - compress a text file and decompress,
- **scp** - send a text file to remote server,
- **ngnix** - serve static content,
- **mongoDB** - insert, select, delete a database entry,
- **ghostscript** - browse an academic paper.

The first four utilities represent three different types of programs on system: *tar* is I/O bounded; *gzip* and *bzip* are CPU intensive; and *scp* is a mixed case. They are popularly used as evaluation for some existing DIFT acceleration work [35, 36, 59]. Applications *gzip*, *nginx*, *mongoDB*, *ghostscript* are server and desktop tasks which are used in JetStream [29], for performance evaluation. We report the mean value of five trials in our experiments.

Table 1: Summary of tested software vulnerabilities. **#Instr.** is the number of instructions from taint sources towards taint sinks of ground-truth in every CVE execution trace.

ID	Program	Vulnerability	#Instr.	CVE ID
1	Nginx	Stack Overflow	146,971	CVE-2013-2028
2	Sndfile-deinterleave	Stack Overflow	40,443	CVE-2018-19432
3	Readelf	Heap Overflow	259,332	CVE-2019-9077
4	Sndfile-convert	Out-of-bound Read	21,424	CVE-2017-14245
5	Thumbnail	Out-of-bounds Write	11,181,696	CVE-2014-8128
6	Eu-ranlib	Divide by Zero	32,935	CVE-2018-18521
7	Size	Stack Overflow	6,030	CVE-2014-9939
8	Thttpd	Information Leakage	17,873	CVE-2009-4491
9	Libsolv	Illegal Address Access	173,606	CVE-2018-20534
10	Mime-parse	Null Pointer Dereference	3,907	CVE-2017-8825
11	Objdump	Null Pointer Dereference	315,580	CVE-2017-17123
12	Cflow	Use-after-Free	5,070,298	CVE-2020-23856
13	Mp42aac	Null Pointer Dereference	146,371	CVE-2020-23912
14	Pngout	Integer Overflow	4,266	CVE-2020-29384
15	Nm-new	Heap Overflow	221,458	CVE-2021-20284

Table 2: Performance of FLOWMATRIX DIFT query and construction, compared with the performance of the CPU baseline for each trace. **FM-Sum** is the summarization time during query tree construction in seconds. **FM-Query** is the query response time in seconds. **CPU-Base** is the propagation time for the CPU-based baseline DIFT tool in seconds.

ID	Program	FM-Sum	FM-Query	CPU-Base
1	Nginx	5.22	0.034	27.57
2	Sndfile-deinterleave	0.93	0.016	6.86
3	Readelf	8.98	0.097	40.21
4	Sndfile-convert	0.49	0.057	3.08
5	Thumbnail	401.39	0.751	1686.26
6	Eu-ranlib	0.61	0.007	4.55
7	Size	0.15	0.001	0.81
8	Thttpd	0.57	0.009	5.18
9	Libsolv	8.31	0.105	68.05
10	Mime-parse	0.09	0.001	0.55
11	Objdump	13.2	0.133	52.78
12	Cflow	187.1	0.469	720.11
13	Mp42aac	7.84	0.266	19.18
14	Pngout	0.10	0.002	0.71
15	Nm-new	9.29	0.115	34.19

6.2 Performance of DIFT Query

We use 15 CVEs listed in Table 1 and seven general tasks to quantify the efficiency and the throughput of FLOWMATRIX DIFT query. We build query tree on top of the range from ground-truth source to sink for each trace to compare with other tools on the same task scales [60]. Then queries are performed by randomly selecting five pairs of common sources (input system calls) and common destinations (output system calls, *rip* register at indirect calls and returns) in security tasks. The average response time is reported in Table 2 and Table 3.

Summarization Performance in Tree Construction. The average speed of FLOWMATRIX summarization is about preparing 54,766 vertices on a query tree per second. For most execution traces of CVEs, the total time costs for summarization are within 10 seconds. For the largest case, `Thumbnail` with 11,181,696 instructions, FLOWMATRIX DIFT query engine efficiently constructs the query tree within 401 seconds. For some heavy application cases (e.g., `mongoDB` and `ghostscript`), if an analyst wishes to compute a query tree for the whole execution trace, which rarely happens in real-world analysis, our tool finished data flow summarization

Table 3: Performance of FLOWMATRIX DIFT query and summarization, compared with Libdft. **Tracing** is the run time of our tracer in seconds. **FM-DFs** is the number of data flows carried by FLOWMATRIX during query tree construction. **FM-TP** is the throughput of data flows per second in FLOWMATRIX query tree construction. **Libdft-Taint** is the run time of Libdft in seconds. **Libdft-DFs** is the number of data flows tracked by Libdft. **Libdft-TP** is the throughput of tracked data flows per second in Libdft.

ID	Program	Tracing	FM-DFs	FM-TP	Libdft-Taint	Libdft-DFs	Libdft-TP
1	Nginx	1.04	21,249,714	4,070,826	3.20	27,306	12,136
2	Sndfile-deinterleave	0.49	5,929,082	4,070,826	1.13	9,655	14,199
3	Readelf	0.42	44,015,827	4,901,540	1.43	9,605	10,218
4	Sndfile-convert	0.35	3,631,270	7,410,755	1.11	2,602	4,066
5	Thumbnail	4.83	1,593,853,705	3,970,836	25.29	105,256	4,160
6	Eu-ranlib	0.44	4,482,776	7,348,813	1.29	11,848	15,190
7	Size	0.48	688,279	4,588,527	1.47	985	1,159
8	Thttpd	0.26	2,978,403	5,225,269	0.98	3,554	5,606
9	Libsolv	0.44	34,120,541	4,105,962	1.14	8,028	12,743
10	Mime-parse	0.35	400,299	4,447,767	1.16	207	309
11	Objdump	0.60	52,226,202	3,956,530	1.69	20,428	20,226
12	Cflow	0.70	832,777,490	4,450,975	1.90	37,764	30,211
13	Mp42aac	0.48	27,766,273	3,541,617	1.45	27,731	31,512
14	Pngout	0.28	769,360	7,693,600	0.87	4,577	8,975
15	Nm-new	0.57	38,608,796	7,693,600	3.67	119,820	39,545

Table 4: The performance DIFT query on traces of seven general Linux applications. **#Instr.** shows the size of program execution traces. **FM-Query** shows the average response time of random queries. Those ones with (*) denote that its query tree vertices have been reduced for easy storage and computation. **FM-Sum** is the summarization time in FLOWMATRIX query tree construction. **FM-DFs** is the number of data flows carried in each tree construction.

Program	#Instr.	FM-Query	FM-Sum	FM-DFs	FM-TP
tar-archive	348,251	0.088	17.2	51,559,046	2,997,619
tar-untar	300,901	0.090	10.4	43,040,613	4,138,520
bzip2-compress	2,704,162	0.238	105.7	400,769,952	3,791,579
bzip2-decompress	537,937	0.127	19.6	80,392,946	4,101,681
gzip2-compress	355,531	0.095	16.2	42,062,045	2,596,423
gzip2-decompress	186,904	0.066	6.7	26,083,421	3,893,048
scp	153,290	0.049	5.4	20,682,643	3,830,119
nginx	1,616,050	0.153	52.7	205,554,578	3,900,466
mongodb	202,003,689	1.974*	9390.3	23,363,491,058	2,488,045
ghostscript	138,985,481	1.811*	5772.7	16,246,131,904	2,814,304

in query tree construction within 9,390 seconds and 5,772 seconds, respectively. For real-world analysis, in most cases, only certain parts of the program execution are interesting and require further DIFT query analysis. Thus, tree size can be further reduced, as well as the tree construction time. Note that FLOWMATRIX is parallelable by design as all instructions are handled in the same operation. Table 4 shows that FLOWMATRIX's performance is positively correlated with the trace length, which indicates FLOWMATRIX scales well with resources proportional to the trace size. Also note that constructing query trees is a one-time effort.

Query Response Time. The runtime for tree construction and query responding is presented in Table 3 and Table 4 for two dataset respectively. The overall runtime of DIFT Query ($\leq$ 0.5 s for most queries, and $\leq$ 1 s for all queries except `mongodb` and `ghostscript`) is substantially smaller than the CPU-based and GPU-assisted taint propagation. For extreme cases with 202 million instructions, our tool manages to answer a DIFT query in less than 2s on average. This is expected because intermediate matrices for queries are precomputed in the query tree construction, resulting in far fewer matrix multiplication operations required in DITF queries.

For cases of `mongodb` and `ghostscript`, by estimation, they require couples of TBs to store the full query tree. Thus, during the construction, we do not store the lowest 10 non-leaf levels of query trees on disk. This results in 1023 times fewer vertices on query trees. As a result of query tree reduction, more matrix multiplications are needed in query. Besides this trade-off, another factor is the richness of dataflows in vertices on trees. The complexity of sparse matrix multiplication is related to matrix's NNZ. Those two factors lead to a longer query time compared with other traces.

Comparison with CPU-based DIFT Baseline. We compare our FLOWMATRIX DIFT query tool with our CPU-based DIFT baseline. The baseline tool takes the same data flow rules but performs less computation: instead of matrix-matrix multiplication, it multiples matrices with a taint map vector on CPU, which is the exact propagation logic for most taint analysis tools. Note that matrix-matrix multiplication is in cubic complexity, in theory, while a vector-matrix multiplication is in square complexity.

Despite heavier computation, Table 2 shows that FLOWMATRIX DIFT summarization still outperforms CPU-based DIFT baseline for around 5.6 times. The performance improvement mainly credits to FLOWMATRIX's ability to propagate data flows in parallel on thousands of GPU cores. In the contrast, CPU baseline has to propagate taint tags from the taint map one by one. We also verify that FLOWMATRIX have the same result as the baseline taking the TaintInduce rules.

6.3 Comparison with Other Data Flow Analysis Solutions

Data Flow Tracking Throughput. To evaluate the ability of FLOWMATRIX query tool, we evaluate its data flow throughput per second. Table 3 presents the numbers of tracked data flows and data flow throughput of FLOWMATRIX summarization in security tasks. On average, FLOWMATRIX summarization achieves a throughput of 5,082,955. Its best result reaches 7,693,600 data flows per second when summarizing the trace of `Pngout`. Notice that our query tool always achieves a higher throughput on small traces than on large ones. The main reason is that matrices in small traces contain fewer data flows to be summarized.

Comparison with LibDFT. LibDFT is one of the state-of-the-art dynamic taint analysis tools widely used in security tasks [61]. Different from our after-the-fact approach, LibDFT is an online instrumentation-based taint tracking tool. For fairness, we present the performance overhead of our dynamic instruction tracer, which writes instruction traces to disk, as a reference for the performance comparison of LibDFT. In general, LibDFT is able to achieve reasonable performance overhead because of limited propagated taint

tags. Thus, we compare the data flow tracking through-puts of FLOWMATRIX with LibDFT to show the power of our tool. Tracked data flows by LibDFT are counted according to the number of propagated taint tags during execution. In our experiment, LibDFT achieves 9,905 data flow through-puts on average, which is only 0.19% of the through-puts in FLOWMATRIX summarization on average. LibDFT's best result reaches 39,545 data flows per second on program `Nm-new`, which remains 0.5% through-puts compared with the case `Pngout` with most through-puts in FLOWMATRIX summarization.

Comparison with JetStream. JetStream is a state-of-the-art DIFT analysis tool supporting DIFT queries based on a server cluster. FLOWMATRIX is fundamentally distinct from JetStream: In terms of acceleration, FLOWMATRIX gain performance speedup by deploying a matrix representation and enabling GPUs as co-processors; JetStream, in contrast, parallels analysis on a cluster of servers. Moreover, FLOWMATRIX is orthogonal to JetStream. That is, FLOWMATRIX can be deployed on servers to further speed up local DIFT computation for each epoch in JetStream.

We compare with JetStream indirectly through a few common Linux tasks. According to the reported numbers in JetStream paper, JetStream is able to answer a DIFT query for the first time on `mongodb`, the largest test case for JetStream and FLOWMATRIX as well, within around 32 seconds with 76,042,962 data flow dependencies. The throughput is around 2,376,342 data flows per-second, with 128 CPU cores. By contrast, FLOWMATRIX DIFT summarization achieves a throughput of 2,488,045 also on the task case `mongodb`, slightly larger than the throughput of JetStream. However, the worst case is `gzip`, where it is reported that JetStream achieves 9,628,096 dependency throughputs while FLOWMATRIX summarization only reaches 2,596,423. In other common Linux programs, FLOWMATRIX outperforms JetStream in throughput by 2,814,304 over 2,097,464 in `ghostscript`. In general, the data flow throughput of FLOWMATRIX is comparable with JetStream, which works with 128 CPU cores.

6.4 Scalability and Bottleneck in Tree Construction

We examine the scalability in tree construction. Then we study the bottleneck and discuss how these bottlenecks can be further addressed.

Scalability. We further evaluate the scalability of FLOWMATRIX query tree construction. Figure 6 shows the speedup of performing query tree construction for each real-world application benchmark on a log-log scale as we vary the number of processes from 1 to 16. Results are normalized to evaluate a construction on a single core; the black diagonal line shows ideal speedup. Overall, multi-processing accelerates DIFT of FLOWMATRIX tree construction by 4-5.5x with a mean of 4.8x using eight processes. By examining the GPU usage,

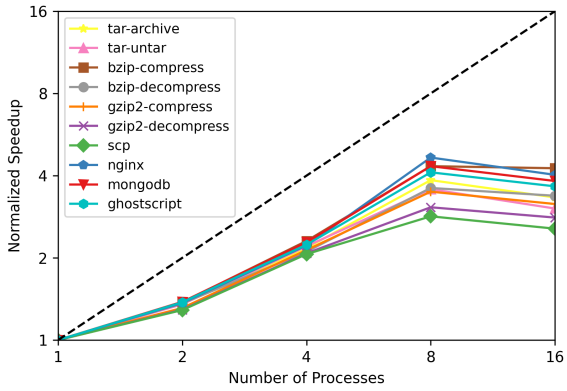


Figure 6: FLOWMATRIX Query Tree Construction Scalability with one GPU

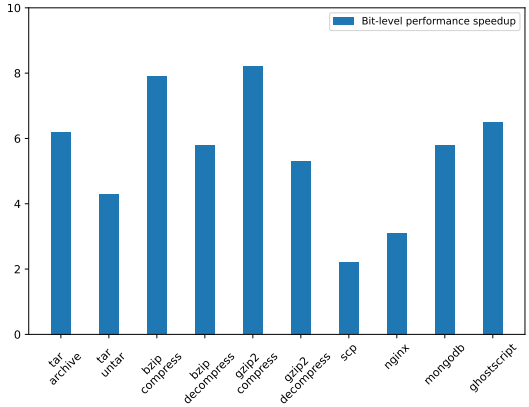


Figure 7: FLOWMATRIX Byte-level performance improvement over bit-level.

we find that a single process only uses around 20%-25% of GPU usage on average; the GPU utilization continues to be above 90% when the number of processes reaches 8 in our experiments. With more GPUs available, FLOWMATRIX is expected to scale up and gain further speedup.

Granularity. Although FLOWMATRIX tracks bit-level data flows by default for accuracy, we provide users with options to track data flows at a more efficient byte level. We evaluate how would a more coarse-grained tracking level would affect our performance shown in Figure 7. On average, data flows are more complicated at the bit level, and the total number reaches 21.8 times more than byte level; the byte-level summarization is 5.5 times faster than bit-level. Among all datasets, `gzip-compress` is most affected by granularity change, which our tool achieves 8.2 times speed up when switching to the byte level. In contrast, `scp` is least sensitive to granularity. The performance only increases by 2.2 times at the byte level.

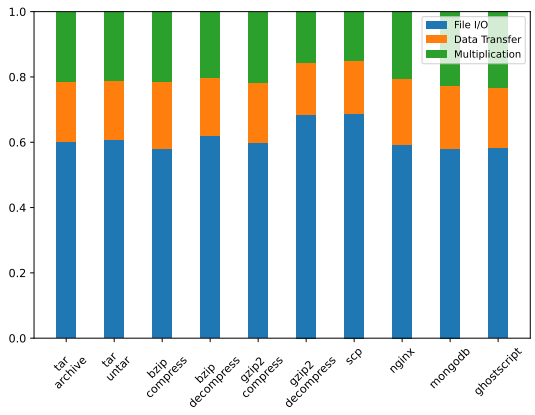


Figure 8: FLOWMATRIX Query Tree Construction Scalability

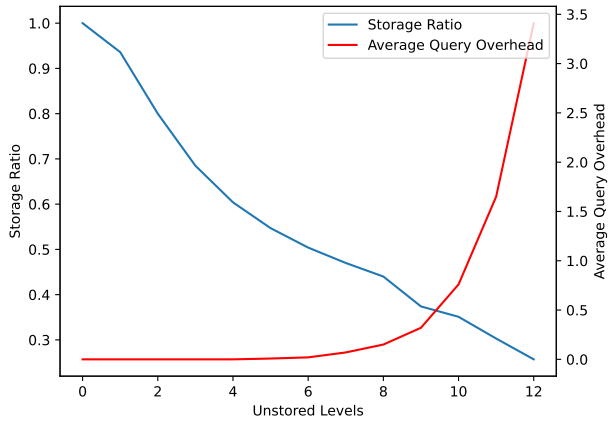


Figure 9: Trade-off between query time and storage size in the optimization of reduced stored levels.

Bottleneck. We next examine the results of FLOWMATRIX query tree construction. Figure 8 shows the stacked bar graphs for each real-world application benchmark at eight processes.

The main bottleneck is file I/O, which takes over 50% of execution time. The reason is that we have reached the upper-bound speed of a single process within a single CPU core. Our tool needs to first load rules into memory from disks and then transfer rules to GPU memory sequentially. In this process, the CPU plays a critical role in scheduling. By increasing the number of processes to eight, we assign more CPU cores to perform file I/O in parallel, increasing all benchmarks' performance. A further improvement would be bypassing CPU and memory, requiring new hardware. For example, in the new generation Nvidia GPU, a technique RTX IO [62] can load data directly from SSD and send it to GPU via PCIe. By adopting such a technique, the performance of FLOWMATRIX tree construction can be further improved.

Query Tree Storage Overhead and Trade-off. In our experiment, the space cost of pre-computed matrices for the DIFT query is proportional to the number of instructions. For those cases where we enable the most efficient DIFT queries, the storage overhead is around 0.325GB per 100k instructions. For most CVE cases within fewer than one million instructions, the storage size is smaller than 10 GB. Those are the cases where we prioritize DIFT query efficiency. In practice, FLOWMATRIX engine can specify the height of the query tree for construction according to particular scenarios. The height can be decreased to maintain low storage overhead. In our largest scenario `mongodb` with over 200 million instructions, by reducing the tree height of 6, we manage to store the query tree within 241.7GB in the disk. We evaluate this trade-off on `mongodb` shown in Figure 9 with the query in the worst case. In general, cutting the lowest six levels or fewer has limited affection to FLOWMATRIX DIFT query usage as at most our tool needs to summarize two sub-trees of the height of 6, but it reduces around 50% storage size in the database. The query overhead grows with more levels not stored on a query tree. When 12 levels are not stored, the storage size ratio drops to 25.7% while the query overhead is non-trivial anymore, reaching 1.65 seconds. In our experiment, we choose to remove ten levels from the query tree for `mongodb` and `ghostscript` to achieve rapid query and acceptable storage overhead. Users are provided with options to balance the query response time and the storage size depending on their own scenarios. Within a large task scale, removing more levels from the query tree would be a good choice to save disk space.

6.5 Case Study

Handling Implicit Flow. We reproduce a case study of implicit control dependency in a popular tiny program, TinyXML [63], which is also reported in Neutaint [64]. As shown in List 1, `p` is a input buffer which stores program input. At line 9, `ele->ClosingType()` gets the member variable `_closingType` and use that variable to determine the program branching behavior. At line 6, `_closingType` can be modified to a constant value (`CLOSING=2`) when there is a special character in the input buffer. As `_closingType` is control dependent but not directly data dependent on input buffer `p`, most DIFT tools fail to track the data flows from input buffer to `_closingType`.

FLOWMATRIX solves this problem by implicit flow augment. List 2 shows a fragment of the execution trace corresponding to the source code in List 1. Register `al` in instruction `cmp al, 0x2f` contains a character from input buffer `p`. On the branch where the `cmp` instruction determines, the instruction `mov dword ptr [rax + 0x68], 2` sets value `CLOSING` to `_closingType`. Later, `ClosingType` is called for the second `if` statement. To analyze whether this `if` statement has a dependency, we query the data flow in backward analysis mode, which queries all possible sources that affect the instruction `je 0x57` in the list. As expected, without implicit flow augment, FLOWMATRIX DIFT query responds

with results of no dependencies. Thus, we use command line prompt to add information flow from `al` to a virtual variable at the first instruction in the list, and add flows from newly added virtual variables to memory cell in the `mov` instruction at the 4-th line. At last, we perform the same query again, and at this time, FLOWMATRIX DIFT query replies that the 1649-th byte from the 8-th `read` system call is the only source that determines the conditional jump instruction `je 0x57`. A forward impact analysis query from the `read` system call also confirms its influence on this instruction and the buffer `p` as well.

```
1  // tinyxml2/tinyxml2.cpp:2044
2  if ( *p == '/' ) {
3     // Implicit control flow dependency
4     ele->_closingType = CLOSING;
5     ++p;
6  }
7  char* XMLNode::ParseDeep(...) {
8     // Broken information flows!
9     if (ele->ClosingType() == CLOSING) {
10       ...
11    }
12 }
```

Listing 1: Source code of implicit control dependency in `TinyXML`.

```
2000459 cmp al, 0x2f ; '/'
2000460 jne 0xe ; Not jumped
2000461 mov rax, qword ptr [rbp - 8]
2000462 mov dword ptr [rax + 0x68], 2
...
2001448 call <ClosingType>
2001453 mov eax, dword ptr [rax + 0x68]
2001455 ret
2001456 cmp eax, 2 ; ==CLOSING?
2001457 je 0x57
```

Listing 2: Part of instructions of implicit control dependency in `TinyXML`. The first column shows instruction index in the program trace.

Misconfiguration Diagnosis. We also reproduce a path traversal misconfiguration on Nginx-1.11.3 [65], an execution trace with 3,899,129 instructions and 822 system calls. File `/etc/passwd` was leaked out through a Nginx-1.11.3 server. We answered three questions used DIFT query: the channel used to leak `/etc/passwd`, (2) the root cause of this leakage, (3) the impact besides the detected leak.

7 Discussion

FLOWMATRIX scales up data flow queries significantly on an offline mode. By taking an execution trace, FLOWMATRIX renders information flow dependencies into linear relationships. Such a matrix representation may also help online dynamic taint tracking with GPUs, which we leave as future work. One challenge could be optimizing data transfer latency. In our approach, fully making use of GPU computation power is more important than low data transfer latency. However, it is the opposite case in online tainting. When PCIe transfers matrices, the CPU may have already executed thousands of instructions. Another challenge is regarding taint checking. Online taint tools require to timely detect security policy violation. However, GPUs are not designed for quick data value checking operations which would lead to alert hysteresis.

Our approach provides low-cost solution for adding implicit flows to analyzed information flows. When facing an under-tainting case, we do not make discussion of policy choice for users, but provide interactive query for debug and information flow augment as an efficient fix after diagnosis. Thus, our solution can be integrated with other work such as MITOS [50] which focuses on the decision problem for indirect flows to automatically fix broken information flows. We leave this as future work.

FLOWMATRIX supports two levels of data flow tracking granularity, bit level and byte level. Bit-level data flow tracking is more precise but heavier. Sometimes it is necessary to perform the analysis at bit-level where the data variables are single bit [66], including flags in *eflags* register on x86 and bit masking. Users may switch the granularity of FLOWMATRIX depending on their own scenarios.

Threats to Validity First, the soundness of FLOWMATRIX depends on TaintInduce [43], which has extensively evaluated soundness and precision of its taint rules compared to existing taint engines. To eliminate the potential error from GPU implementation, we also verified the correctness of FLOWMATRIX results by cross-checking with our CPU-based DIFT tool using the same TaintInduce rules. Second, FLOWMATRIX is also affected by under-tainting / over-tainting. Under-tainting and over-tainting cases are consequences of inaccurate taint policy – for example, whether to track data flows for *eflags* register. To address this problem, FLOWMATRIX provides the analysts with options of information flow augment for fine-tuning data flow propagation policy.

8 Related Work

FLOWMATRIX is a novel representation of taint rules, which enables the first system to use GPUs as co-processors. Also, it is the first system to efficiently support DIFT query without a complex backend server infrastructure. In this section, we discuss the closely related work in speed up DIFT rule processing.

8.1 Information Flow Rule Representation

Prior work in taint analysis, such as TaintCheck [1], LibDFT [45], commonly implements hand-written taint propagation rules within advanced program languages and only tracks direct data flow impacts.

Several prior works have proposed different taint rule representations other than direct implementation in advanced languages. Jee et al. [41] proposed *Taint Flow Algebra* to summarize the semantics of taint logic for basic blocks. Nevertheless, the scale of summarization is not un-limited. TaintInduce [43] learns dataflows by executing instruction and mutating inputs. FLOWMATRIX 's taint rule is derived from this work. We are the first one to identify the linearity in taint propagation and lift taint rules generated by TaintInduce to matrices. Proximal gradient analysis has also been adopted to track data flows with gradients over operations in programs [44]. This approach is orthogonal to TaintInduce and can be deployed as our taint rule generator by transforming gradients into matrices or a seizes of GPU operations.

A common problem in taint analysis is implicit flow tracking [28]. There are solutions using dependency-based taint concept to address this problem. NeuTaint [64] identifies taint relationship via neural program embeddings at function scale. Bao et al. [52] tracks strict control flow dependencies by computing the lineage of variable constraints. In contrast, FLOWMATRIX allows users to specify customized DIFT policy to track implicit flows.

8.2 Query-style DIFT

FLOWMATRIX DIFT query is an after-the-fact analysis that supports multiple DIFT queries. Most DIFT parallelization work has focused on live analysis, which only supports a single pre-defined DIFT query. The main related work of after-the-fact DIFT query is JetStream [29], a system to support DIFT query by parallelizing DIFT across a cluster. It records the program execution first and then partitions the replay and DIFT analysis into epochs. This approach is orthogonal to ours, as FLOWMATRIX can be deployed to a cluster of GPU servers. The forward propagation in JetStream can be replaced by our GPU-assisted DIFT propagation to achieve further speedup.

8.3 Hardware-assisted DIFT

Extensive literature has explored how to speedup DIFT itself with special hardware by offloading the taint operations to co-processors. Ruwase et al. [31] implement an algorithm in the context of a *Log-Based Architectures* (LBA) system which logs a program trace and delivers it to other processors. Nagarajan et al. [32] use a separate core of a chip multiprocessor (CMP) to track taint. Similar to those specialized architectures for dynamic information flow tracking, the requirement of special hardware prevents existing hardware-assisted approaches from being adopted using commodity hardware. To our best knowledge, there are no prior studies using GPUs as co-processors for DIFT. GPUs are getting more and more popular, and have become common commodity hardware. Moreover, many commodity CPUs have integrated graphics processors, such as Intel Graphics, which also support GPGPU development and thus can be used as FLOWMATRIX's co-processors.

FLOWMATRIX is the first one adopting GPUs to speed up DIFT analysis. There are other prior work that also uses GPUs in security tasks. Yu et al. [67] deploy GPUs for fast Android App vetting by speeding up worklist algorithm for IDFG construction. Vasiliadis et al. [68] reveal the abilities of GPUs to accelerate microarchitectural attacks. Velea et al. [69] propose an CPU/GPU hybrid improvement to accelerate string matching for malware signature detection with low power consumption. Mendez-Lojo et al. [70] introduce a GPU-based point-to analysis and outperforms a sequential CPU implementation for 7 times in performance.

9 Conclusion

The performance overhead of dynamic information flow analysis has long been the problem limiting its applications, which is prohibitively expensive for query-style applications. In this paper, we analyze the recently developed dependency-based information flow propagation rules and recognize the linearity of DIFT operations in offline dynamic information flow analysis. Based on this finding, we present a novel matrix-based representation, FLOWMATRIX, which enables using GPUs as co-processors for DIFT operations. We further build a system that supports information flow queries efficiently to assist analysts in interactively querying data flow properties of programs. Our evaluation shows that FLOWMATRIX scales up data flow queries significantly and gains tremendously speedup from GPU. We also study the performance improvement and bottlenecks and demonstrate the applications in taint analysis diagnosis and configuration file diagnosis.

Acknowledgment

We thank our shepherd, Jiang Ming, and the anonymous reviewers for their valuable comments. We also thank Jiahao Liu for his help with experiments. Some of the experiments were conducted using the infrastructure of National Cybersecurity R&D Laboratory, Singapore. This research is supported by the National Research Foundation, Singapore under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative, by the National Research Foundation, Singapore under its NSoE DeST-SCI programme (Grant No. NSoE_DeST-SCI2019-0006), and by Ministry of Education of Singapore under the project "Algorithmic Advances For Program Fuzzing" (MOE-T2EP20220-0014). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of National Research Foundation, Singapore.

References

[1] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, 2005.

[2] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS*, 2007.

[3] Prateek Saxena, R Sekar, and Varun Puranik. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *CGO*, 2008.

[4] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *ISSTA*, 2007.

[5] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *ICSE*, 2009.

[6] Min Gyung Kang, Stephen Mccamant, and Pongsin Poosankam. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *NDSS*, 2011.

[7] R Sekar. An efficient black-box technique for defeating web application attacks. In *NDSS*, 2009.

[8] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qaualifiers. In *USENIX Security Symposium*, 2001.

[9] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, Omri Weisman, Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In *PLDI*, 2009.

[10] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE S&P*, 2010.

[11] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *ACM CCS*, 2007.

[12] Wei Xu, Sandeep Bhatkar, and Ramachandran Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security Symposium*, 2006.

[13] Jingfei Kong, Cliff C Zou, and Huiyang Zhou. Improving software security via runtime instruction-level taint checking. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, 2006.

[14] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO*, 2006.

[15] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *International Workshop on Recent Advances in Intrusion Detection*, 2005.

[16] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *ACM CCS*, 2017.

[17] Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee. Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking. In *USENIX Security Symposium*, 2018.

[18] Jun Zeng, Zheng Leong Chua, Yinfang Chen, Kaihang Ji, Zhenkai Liang, and Jian Mao. Watson: Abstracting behaviors from audit logs via aggregation of contextual semantics. In *NDSS*, 2021.

[19] Jun Zeng, Xiang Wang, Jiahao Liu, Yinfang Chen, Zhenkai Liang, Tat-Seng Chua, and Zheng Leong Chua. Shadewatcher: Recommendation-guided cyber threat analysis using system audit records. In *IEEE S&P*, 2022.

[20] Neil Vachharajani, Matthew J Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A Blome, George A Reis, Manish Vachharajani, and David I August. Rifle: An architectural framework for user-centric information-flow security. In *MICRO*, 2004.

[21] David Yu Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. Tainteraser: Protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review*, 2011.

[22] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *USENIX Security Symposium*, 2004.

[23] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI*, 2010.

[24] Julian Schutte, Dennis Titze, and J.M. M.De Fuentes. AppCaulk: Data leak prevention by injecting targeted taint tracking into android apps. In *TrustCom*, 2015.

[25] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol specification extraction. In *IEEE S&P*, 2009.

[26] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI*, 2010.

[27] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, 2012.

[28] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE S&P*, 2010.

[29] Andrew Quinn, David Devecsery, Peter M Chen, and Jason Flinn. Jetstream: Cluster-scale parallelization of information flow queries. In *OSDI*, 2016.

[30] Edmund B Nightingale, Daniel Peek, Peter M Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In *ACM Sigplan Notices*, 2008.

[31] Olatunji Ruwase, Phillip B Gibbons, Todd C Mowry, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, and Michael Ryan. Parallelizing dynamic information flow tracking. In *the 20th Annual Symposium on Parallelism in Algorithms and Architectures*, 2008.

[32] Vijay Nagarajan, Ho-Seop Kim, Youfeng Wu, and Rajiv Gupta. Dynamic information flow tracking on multicores. In *the Workshop on Interaction Between Compilers and Computer Architectures*, 2008.

[33] Andrey Ermolinskiy, Sachin Katti, Scott Shenker, L Fowler, and Murphy McCauley. Towards practical taint tracking. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-92*, 2010.

[34] Kangkook Jee, Vasileios P Kemerlis, Angelos D Keromytis, and Georgios Portokalidis. Shadowreplica: efficient parallelization of dynamic data flow tracking. In *ACM CCS*, 2013.

[35] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. Taintpipe: pipelined symbolic taint analysis. In *USENIX Security Symposium*, 2015.

[36] Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu. Straighttaint: Decoupled offline symbolic taint analysis. In *ASE*, 2016.

[37] Jim Chow, Tal Garfinkel, and Peter M Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX ATC*, 2008.

[38] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. Paranoid android: versatile protection for smartphones. In *ACM ACSAC*, 2010.

[39] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The world's fastest taint tracker. In *RAID*, 2011.

[40] Ali Davanian, Zhenxiao Qi, Yu Qu, and Heng Yin. Decaf++: Elastic whole-system dynamic taint analysis. In *RAID*, 2019.

[41] Kangkook Jee, Georgios Portokalidis, Vasileios P Kemerlis, Soumyadeep Ghosh, David I August, and Angelos D Keromytis. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *NDSS*, 2012.

[42] Olatunji Ruwase, Shimin Chen, Phillip B Gibbons, and Todd C Mowry. Decoupled lifeguards: enabling path optimizations for dynamic correctness checking tools. In *ACM Sigplan Notices*, 2010.

[43] Zheng Leong Chua, Yanhao Wang, Teodora Baluta, Prateek Saxena, Zhenkai Liang, and Purui Su. One engine to serve'em all: Inferring taint rules without architectural semantics. In *NDSS*, 2019.

[44] Gabriel Ryan, Abhishek Shah, Dongdong She, Koustubha Bhat, and Suman Jana. Fine grained dataflow tracking with proximal gradients. In *USENIX Security Symposium*, 2021.

[45] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *VEE*, 2012.

[46] Heng Yin and Dawn Song. Temu: Binary code analysis via whole-system layered annotative execution. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-3*, 2010.

[47] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In *ISSTA*, 2014.

[48] Wikipedia contributors. Sparse matrix — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Sparse_matrix, 2021. [Online; accessed 20-December-2021].

[49] Steven Dalton, Luke Olson, and Nathan Bell. Optimizing sparse matrix—matrix multiplication for the gpu. *ACM Transactions on Mathematical Software (TOMS)*, 41(4):1–20, 2015.

[50] Nikolaos Sapountzis, Ruimin Sun, Xuetao Wei, Yier Jin, Jedidiah Crandall, and Daniela Oliveira. Mitos: Optimal decisioning for the indirect flow propagation dilemma in dynamic information flow tracking systems. In *ICDCS*, 2020.

[51] Hong Hu, Zheng Leong Chua, Zhenkai Liang, and Prateek Saxena. Identifying arbitrary memory access vulnerabilities in privilege-separated software. In *ESORICS*, 2015.

[52] Tao Bao, Yunhui Zheng, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Strict control dependence and its effect on dynamic information flow analyses. In *ISSTA*, 2010.

[53] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.

[54] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*. 2019.

[55] NVIDIA. cusparse. https://docs.nvidia.com/cuda/cusparse/.

[56] Matthias Fey. Pytorchsparse. https://github.com/rusty1s/pytorch_sparse, 2022.

[57] National vulnerability database. https://nvd.nist.gov/.

[58] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent dynamic instrumentation. *ACM SIGPLAN Notices*, 47(7):133–144, 2012.

[59] John Galea and Daniel Kroening. The taint rabbit: Optimizing generic taint analysis with dynamic fast path generation. In *ACM AsiaCCS*, 2020.

[60] Erik van der Kouwe, Gernot Heiser, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. Sok: Benchmarking flaws in systems security. In *IEEE EuroS&P*, 2019.

[61] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *IEEE S&P*, 2018.

[62] NVIDIA. Nvidia rtx io: Gpu accelerated storage technology. https://www.nvidia.com/en-sg/geforce/news/rtx-io-gpu-accelerated-storage-technology/.

[63] Tinyxml2. https://github.com/leethomason/tinyxml2. 2022.

[64] Dongdong She, Yizheng Chen, Abhishek Shah, Baishakhi Ray, and Suman Jana. Neutaint: Efficient dynamic taint analysis with neural networks. In *IEEE S&P*, 2020.

[65] Cwe-22: Improper limitation of a pathname to a restricted directory ('path traversal'). https://cwe.mitre.org/data/definitions/22.html, 2021.

[66] Babak Yadegari and Saumya Debray. Bit-level taint analysis. In *IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, 2014.

[67] Xiaodong Yu, Fengguo Wei, Xinming Ou, Michela Becchi, Tekin Bicer, and Danfeng Daphne Yao. Gpu-based static data-flow analysis for fast and scalable android app vetting. In *IEEE IPDPS*, 2020.

[68] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand pwning unit: Accelerating microarchitectural attacks with the gpu. In *IEEE S&P*, 2018.

[69] Radu Velea and Ştefan Drăgan. Cpu/gpu hybrid detection for malware signatures. In *2017 International Conference on Computer and Applications (ICCA)*. IEEE, 2017.

[70] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. A gpu implementation of inclusion-based points-to analysis. *ACM SIGPLAN Notices*, 2012.

[71] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *ACM CCS*, 2013.

A Appendix

A.1 Information Flow Rules

Figure 10 shows the taint rules to track information flow for the instruction `add` in three different taint engines, `TEMU` [46], `libdft` [45] and `DECAF` [47]. `libdft` implements taint propagation rules as stand alone propagation functions that are instrumented during execution while QEMU-based `TEMU` and `DECAF` implement the taint rules in the emulator by modifying the emulated instructions to propagate taint. This results

```
void taint_parallel_compute(
    shad, dest, opcode, ...)
  {
  if (opcode == llvm::
    Instruction::Or) {
    cb_mask_out.cb_mask =
     (cb_mask_1.zero_mask &
    cb_mask_2.cb_mask) |
     (cb_mask_2.zero_mask &
    cb_mask_1.cb_mask);
  }
  write_cb_masks(shad, dest,
    cb_mask_out,...);
  ...
  }
```

(a) Panda

```
void r2r_binary_opl(dst, src,
    ...) {
  threadctx->vcpu.gpr[dst]|=
    threadctx->vcpu.gpr[src];
}
void ins_inspect(INS ins) {
  ...
  switch (ins_indx) {
    case XED_ICLASS_OR:
      INS_InsertCall(
      r2r_binary_opl, REG32_INDX
      (reg_dst), REG32_INDX(
      reg_src), ...);
  }
}
```

(b) Libdft

```
int gen_taintcheck_insn(...){
  switch(opc) {
  case INDEX_op_or_i32:
    /* t0 = arg1 || arg2 */
    tcg_gen_or_i32(t0,a1,a2);
    /* t2 = (t0 != 0)    */
    tcg_gen_movi_i32(t_z,0);
    tcg_gen_setcond_i32(
    TCG_COND_NE,t2,t_z,t0);
    /* res = ~t2         */
    tcg_gen_neg_i32(res,t2);
    break;
    ...
  }
}
```

(c) Decaf

Figure 10: Different implementations of the taint rule of *or* instruction in `Panda`, `libdft` and `DECAF`. `Panda` defines controlled bit mask to calculate which bits are attacker controlled. `libdft` directly performs an *or* operation from *src*'s tag to *dst*'s tag. `DECAF` encodes the propagation logic with tcg IR. Code has been simplified for easier reading.

in multiple, non-interoperable ways of propagating taint between the different approaches. Although the taint propagation semantics for the instruction is the same, the implemented rules are wildly different. `TEMU` defines taint rules for different instructions with different op-codes and implements taint propagation in other functions. `libdft` inlines taint propagation in instrumentation during execution. It is a huge case switch of op-code, decoded by X86 Encoder Decoder (XED), and in each case lays taint propagation. Although `DECAF` is also QEMU-based, similar to `TEMU`, it propagates taint in Tiny Code Generator (TCG) level, which is an intermediate representation (IR). These difference can be seen as the result of ad-hoc, tightly coupled implementation specific paradigm used in the design of these engines. For example the type of statement where taint analysis is used on results in some rules being defined on top of intermediate representations (IR), while others directly on machine instruction set architectures (ISAs). Other than making reusing of existing taint rules next to impossible, these implementation specific designs also prevents a single, unified way of propagating taint.

A.2 Matrix Normalization

However, there is still a gap to fill when multiplying data flow matrices: each data flow matrix is using local variable state and unaware of others' context. For instance, we consider a task to summarize the data flow of `mov rax, [rbp-8]` and `and rcx, rax`. The size of two data flow matrices are both 128×128, but in a different context: the variable state, `[rax, [rbp-8]]`, of the first instruction consists of `rax` (size of 64) and `[rbp-8]` (size of 64), while the second is `[rcx, rax]` with the same length of 128. Thus, before the summarization, there is a need to normalize both data flow matrices within the superset of variable states.

Intuitively, there are two steps in normalization: extending and reordering. Specifically, a $n \times n$ local matrix M has to be first extended to $m \times m$ to match the shape of the normalized matrix by appending zero rows and zero columns. Next, rows and columns in the extended matrix must be reordered to match the order of normalized variable state. For example, if the goal is to normalize the data flow matrix of `and rcx, rax` to a normalized state `[rax, [rbp-8], rcx]`, we can first extend the state to `[rcx, rax, NULL]` and then reordering it to `[rax, NULL([rbp-8]), rcx]`. Let E be an identity matrix I_n concatenated with $m-n$ zero rows. The extension of M can be achieved by pre-multiplied with E and post-multiplied with E^T, where E^T is the transpose of E (i.e., an identity matrix I_n concatenated with $m-n$ zero columns). After matrix extension, the reordering can be divided into elementary operations: row-switching transformations and column-switching transformations. As the extended matrix is also a square matrix of which rows and columns both represent the extended state, the reordering is equivalent to pre-multiply a permutation matrix P and post-multiply its transpose P^T. So far, the production matrix M' is a $m \times m$ matrix where $M' = P(EME^T)P^T$. Let U be the product of $P \times E$, then the above formula can be simplified to $M' = UMU^T$. So far, M' is ready for data flow summarization, except for one thing: it does not inherit data flows from extended variables. Thus, ones have to be placed on the main diagonal for those extended rows and columns and update our formula: $\hat{M} = U(M - I_n)U^T + I_m$.

A.3 Implementation Details

System Call Hooking. Traditional DIFT analysis tools hook system calls, and set taint status for source system calls

or check taint status for sink system calls [4,45]. FLOWMATRIX does not follow the design of taint tag propagation but tracks all information flows in matrices. Our matrix design brings a new challenge in handling system calls. Namely, there is no proper column in matrix can be chosen as the information flow source for an input system call, such as `read`, `recvfrom`, etc. The reason is that, as a system call is for a program to request a service from the kernel, from the point of view of a process, a system call introduces new information flows from "nowhere": they do not originate from any in-process variable. To address this challenge, we need to introduce a new data variable into the matrix for each system call. That is, similar to how FLOWMATRIX handles implicit flows, we extend the matrix with one column and one row representing the system call variable for each system call in trace. For each of them, the appended variable is influenced by all of the system call's arguments and influences the return value (`rax` register). Furthermore, for those input system calls which load data into process, the system call variable influences the loaded buffer; for those output system calls which write data to externals, the written buffer influences the system call variable.

Recall that it is creating a global information flow snapshot for a certain position by adding a new variable into matrix and setting information flows towards it. Then, setting information flows from an extended variable is creating a global information flow source. By combining two types of operations, FLOWMATRIX enables us a rapid n-to-n query among multiple system calls in O(1) time, which has wide applications in sensitive data leakage detection [24], dependency explosion resolution [16, 17] and etc. The rationale is, unlike other data variables, such snapshots and global sources are not limited by variable scopes and can be access from a global view. Specifically, in the matrix of a root vertex on a query tree, its sub-matrix of all system call variables over themselves is a linear map describing dependencies among all system calls. Informally, a non-zero element in the sub-matrix represents there is an information flow from the system call corresponding to its column to another system call corresponding to its row.

Constant Flow Tracking. Traditional DIFT analysis do not track data flows from a constant (e.g., `mov eax, 1`) or a sanitized register (e.g., `xor eax, eax`) [28]. This is reasonable as constant flows are not worth tracking in most cases. However, in some data-flow-based security scenarios, analysts do have the needs to query the impact of a constant. For example, in cryptographic misuse detection, analysts would check whether a constant has been set as the random seed for a cryptographic function [71]. By using traditional DIFT tools, the analysts have to manually modify the tool's code to hook a taint status set function to the place where the constant is introduced and register a taint status check function of arguments when the program calls a cryptographic function. This requires expert knowledge about a DIFT tool and code modification for each scenario. FLOWMATRIX takes into account both situations and provides users with code-free constant flow queries. In common scenarios, we treat instructions of constant flows as empty information flows (operands do not inherit their own data flow). If a query's source is a constant flow instruction, then FLOWMATRIX automatically tracks the information flow from the constant. Specifically, we extend the source instruction's matrix with one row and one column similar to the scenarios in system calls and set data flows from the extended variable to the destination operands in instruction. Then we query the information flow from its next instruction to the destination and multiply the extended matrix with the query result.