MITOS: Optimal Decisioning for the Indirect Flow Propagation Dilemma in Dynamic Information Flow Tracking Systems [*]

paper number: 194

Abstract

Dynamic Information Flow Tracking (DIFT), also called Dynamic Taint Analysis (DTA), is used to keep track of the information as it flows through a program's or system's execution. Specifically, some inputs or data get tainted and then these taint marks (tags) propagate usually at the instruction-level. An open problem in this context, that impedes the widespread application of DIFT in practice, is: *should the tags involved in an indirect flow, e.g., in a control or address dependency, be propagated?* Propagating all these tags, as done for direct flows, leads to overtainting (all taintable objects becomes tainted), while not propagating them leads to undertainting (information flow becomes incomplete). In this paper, we formulate the problem of optimal propagation for indirect flows that (i) analytically investigates the tradeoff between undertainting and overtatinting and (ii) introduces fairness and tag-balancing, which are key properties for the success of modern DIFT that will leverage tags of different types. Towards tackling this decisioning problem that is shown to be NP-hard, we propose MITOS, an iterative algorithm and a set of propagation policies that are based on distributed optimization. We show that MITOS converges to an optimal point flexibly optimizing undertainting, overtainting, tag type importance and fairness. MITOS is scalable, of-low complexity, and can be adopted in most DIFT systems. We implemented and evaluated MITOS in FAROS, an open-source software-based DIFT. Our sensitivity analysis shows that MITOS optimizes the above-mentioned tradeoffs offering insights to this open decisioning problem. Further, MITOS reduced FAROS' time overhead by 40%, FAROS' memory overhead by 10%, and increased FAROS' fingerprint on suspected bytes during an in-memory-only attack by 167%.

Keywords indirect flows, distributed optimization, information flow tracking, in-memory-only attacks

1 Introduction

Dynamic Information Flow Tracking (DIFT), or Dynamic Taint Analysis (DTA), systems operate by tainting various inputs or data of interest with some metadata (called tags) and keeping track of these tags throughout a program or system execution. They operate dynamically without requiring the availability of the source code, which makes them appealing for various types of applications, including forensics analysis, reverse engineering, and security policies. Prior work has attempted to leverage DIFT to better understand attackers' intentions, detect attacks or perform network protocol analysis in different scenarios including software, hardware, and mobile devices [1–6].

Nevertheless, DIFT systems still face open challenges that impede their widespread application in practice. One of these challenges is the dilemma of indirect flow dependency propagation. An indirect flow occurs when information dependent on the program input determines from where and to where information flows. For example, in the code $< a = b + 1 >$, there is a direct flow from b to a, and all DIFT systems would propagate the tag of b to a. However, in the code $< a = 0; \quad if \quad (b == 1) \quad \{a = 1\}; >$, the value of a is dependent on b, meaning that there is an indirect flow from b to a. Not propagating tags in these cases can lead to *undertainting*, where key important information flows are missed. Propagating tags for all indirect flow dependencies leads *overtainting*, where most of the taintable objects in the system (e.g., bytes) become tainted and thus little useful information is acquired.

While previous works have proposed interesting heuristics to tackle the problem, they usually make unrealistic assumptions to modern systems and have several limitations. For example, Panorama [7] relies on a human to manually label which indirect flow dependencies should be propagated. DTA++ [8] or DYTAN [9] rely on offline analysis requiring multiple traces, which does not scale well. RIFLE [10] and GLIFT [11] are based on static analysis, and other works have prohibitive performance overheads [2, 12]. While useful, these techniques can only partially combat the problem.

In this paper, we propose MITOS, an analytical framework composed of a set of policies that optimally tackles the open problem of: *when an indirect flow should be propagated in a DIFT system.* To the best of our knowledge, this is the first work that *analytically* considers this problem, opening new

[*]MITOS, in Greek mythology, was a ball of thread, that Ariadne gave to Theseus to help him escape the labyrinth of Minos kingdom. As MITOS helped Theseus to reversely find his way back to labyrinth's entrance by minimizing his wandering, our framework minimizes the incoherent tag propagations (e.g. of indirect flows), helping illuminating the information flow from a certain output all the way back to the input.

horizons of how the information-flow at the systems level can be modeled and optimized. Specifically, the contributions of this paper are as follows:

(1) We model a framework for the problem of propagating indirect flows dependencies, optimally weighting various tradeoffs such as the undertaining vs. overtainting, and the information flow importance between different tag types.

(2) While the considered problem is NP-hard, we propose a distributed optimization algorithm that: (i) converges to an optimal point, (ii) is of low-complexity, scalable, flexible and, thus, potentially applicable to most DIFT sytems, and (iii) is extendable to direct flows and different cost functions.

(3) We introduce the *fairness* property in the DIFT field, which controls the balancing among the propagations of different tags. It matches information-theoretic intuitions about how tags should be propagated: flipping a coin that has a $50\% - 50\%$ chance of heads-tails carries more information than a coin that is biased in one direction [13]. Similarly, when tag propagation becomes unbalanced towards one tag, every object is tagged and little information is gained.

(4) We implemented MITOS on top of FAROS, an existing software-based DIFT system that performs byte-level tainting [12]. We analyze the complex tradeoffs involved in the indirect flow propagation dilemma e.g., the impact of undertainting vs. overtainting on the Pareto optimal distribution. We showed that MITOS reduced FAROS time overhead by 40%, FAROS memory overhead by 10%, and increased FAROS fingerprint on suspected bytes by 167% during an in-memory-only attack.

The rest of the paper is organized as it follows. Section 2 provides basic background on DIFT and Section 3 discussed MITOS assumptions. Section 4 details the analytical model for the indirect flow propagation problem and its optimal solution. Section 5 discusses MITOS' implementation in an existing DIFT and Section 6 discusses our experimental evaluation and results. Section 7 summarizes MITOS key findings, limitations, and future work. Section 8 presents related work and Section 9 concludes the paper.

2 DIFT - background

Dynamic Information Flow Tracking (DIFT), or Dynamic-Taint Analysis (DTA), a fundamental concept in computer and network security, is a promising method to make systems transparent and enable a wide variety of applications, such as real-time forensics analysis, malware detection and analysis via reverse engineering. The main idea is based on tagging certain inputs or data (tag insertion), and then, propagating these tags as the program or system runs (tag propagation) with the goal of illuminating the flow of information.

Tag insertion is usually a straight-forward process, where the bytes involved in certain system activities are tagged with some metadata. For example, in MINOS [14], one of the first DIFT systems, all data coming from network were tagged with an extra bit indicating whether or not the byte was suspicious. There are two types of tag propagation flows: direct and indirect.

```
char InputString = "This string is tainted";
char OutputString[128];
for (i = 0; i < strlen(InputString); i++)
    OutputString[i] = lookuptable[InputString[i]];
```

Figure 1. Address dependency example.

Direct flow propagations (DFP) come from either copy or computation dependencies. In a copy dependency, a value is copied from one location to another, where a location can be, for example, a byte, a word of memory, or a CPU register. To track this information flow, DIFT systems propagate the tag from the source to the destination. In computation dependencies, tags must be combined. For example, after the computation of a sum between two integer operands, the tag of the result should contain both tags of operands.

Indirect flow propagations (IFP) occur when information dependent on program input determines from where and to where information flows. There are two types of indirect flows: *address* and *control dependencies*, with several examples available in the literature [2, 6]. Figure 1 provides an address dependency example in C that converts an array of tainted input from one format to another using a lookup table. There, as the string `InputString` is tainted, the string `OutputString` should also be tainted, since they carry the same information. To ensure that `OutputString` is properly tainted we check the taintedness of the address used for the load with `LookupTable` as its base, and propagate this taint. This example appears in special handling of ASCII control characters to ASN.1 encodings. Generally, indirect flows are expected to be the rule rather than the exception in modern systems, occurring in operations such as in compression/decompression, encryption/decryption, hashing, switch statements, string manipulations. Indirect flows can create blindspots for practical DIFT analysis or vulnerabilities in security applications e.g., Trojans embedded in PDF documents or attacks that use encryption mechanisms are common, but cannot be tracked without tracking indirect flows.

Propagating all indirect flows can lead to *overtainting*, where most of the objects become tainted and very little can be learned about the information flow. Conversely, not propagating indirect flows can lead to *undertainting*, where important knowledge about the information flow might be lost. While several works have attempted to tackle this dilemma, it still remains open, and constitutes one of the major impedances to the widespread usage of DIFT. The focus of this paper is on optimally tackling this problem.

3 MITOS Assumptions

In this section we describe MITOS operational assumptions.

Tag differentiation. First, MITOS assumes that the DIFT system will leverage an arbitrary number of tag types. For example, it could include: *network* tags (representing bytes coming from network), *file* tags (representing bytes coming from a file), *CR3/process* tags (representing bytes coming from the address space of a process). Throughout this paper we will use the terms CR3 tags and process tags interchangeably. For the sake of presentation, we denote the different tag types as: $t1, t2, \ldots$. Tag differentiation is a promising feature of modern DIFT systems since it allows the capture of information flows from different perspectives [12, 15].

Provenance list. MITOS assumes that for each byte in the main memory, register bank and Ethernet card memory, a *provenance list* of tags accumulated during the system execution. MITOS also assumes that different tag types will have different formats and sizes depending on the type of information they represent; i.e., network, file, process, string, pointer tags. The provenance list, through the set of tags it stores, keeps all information flow history for the lifecycle of a byte in the system. For example, Fig. 2 illustrates the provenance list for the byte representing memory address #7FFFFF8. This byte came from a network source (IP=10.245.44.43), was read as part of the address space of a process (PID 3543), was written into a file (with file ID 14) and then was read as part of an address space of another process (PID 2912).

Provenance list size. The provenance list size is finite, denoted as M_{prov}. For example, $M_{prov} = 10$ means that each byte can keep up to 10 different tags in its provenance list.

Shadow Memory. MITOS assumes that the provenance list of tags for each byte will be stored in a shadow memory, whose implementation will vary according to the DIFT system, e.g., it could be a hashmap or duplicated memory. Also, MITOS needs to keep track of the current level of memory being used to store tags, which we call *memory pollution.*

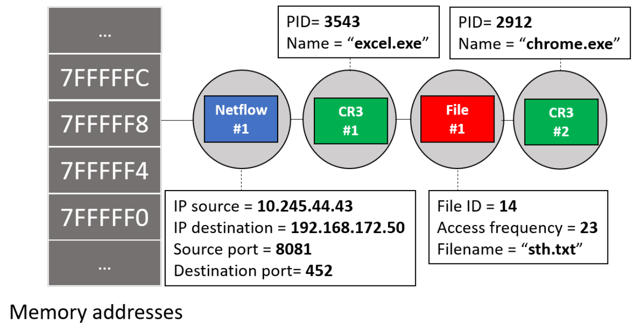


Figure 2. Provenance list for the byte in address #7FFFFF8.

4 MITOS: Optimal Decisioning for Indirect Flow Propagations

This section describes MITOS theoretical framework and derived policies. MITOS' goal is to address the indirect flow propagation dilemma. To that end, we first formulate an optimization problem for that problem (Section 4.1), analytically derive the optimal solution to the problem through Alg. 2 (Section 4.2), and generalize this algorithm to capture direct flows and other objectives (Section 4.3).

4.1 Problem Formulation

We first (A) define the variables that are associated with the indirect flow dilemma corresponding to our *control variables.* Then, (B) we design a new *cost function* that attempts to optimally weight the different tradeoffs involved at the indirect flow propagation. Finally, (C) we define our *optimization problem* (that we will solve in the next Section 4.2). Table 1 summarizes the notation used throughout the section; the indicator (input parameter) refers to the inputs in our model.

(A. Control Variables: n). When the DIFT system is confronted with an indirect flow dependency, it needs to decide whether it is worth propagating that particular tag to one additional byte. For the sake of presentation, let us assume that each particular tag has a unique ID $\{t, i\}$, where $t \in \mathbb{T} = \{t1, t2\}$ indicates the tag type, and $i \in \mathbb{N}^+ = \{1, 2, 3, \ldots\}$ represents an integer that differentiates tags of same type. We now define as $n_{t,i} \in \mathbb{N} = \{0, 1, 2, \ldots\}$ to be the number of bytes whose provenance lists contain the tag with ID $\{t, i\}$. Throughout this paper, we will often refer to $n_{t,i}$ as *the number of copies* of the tag with ID $\{t, i\}$. For instance, if $t1$ is of type *network*, we could have two different network tags each associated with two network connections. Thus, $n_{t1,1}$ would describe the number of copies of the tag $t1, 1$, while $n_{t1,2}$ of the tag $t1, 2$. The control vector $\mathbf{n}$ is:

$$\mathbf{n} = \begin{pmatrix} n_{t1,1} & n_{t1,2} & n_{t1,3}, & \cdots \\ n_{t2,1} & n_{t2,2} & n_{t2,3}, & \cdots \\ \cdots & \cdots & \cdots & \ddots \end{pmatrix} \tag{1}$$

The number of dimensions of $\mathbf{n}$ changes dynamically as the system runs, since new tags are created/deleted, e.g., due to the continuous creation/ termination of processes, network connections etc. This prevents the usage of standard optimization techniques, further complicating the problem.

(B. Cost function: $c_{\alpha,\beta}(\mathbf{n})$). We now define our cost function $c_{\alpha,\beta}(\mathbf{n})$ that dynamically weights the cost of α-fair undertainting and the cost of β-steep overtainting.

$$c_{\alpha,\beta}(\mathbf{n}) = \overbrace{c_{\alpha}^{under}(\mathbf{n})}^{\text{cost of undertainting}} + \underbrace{\tau}_{\text{weight}} \cdot \overbrace{c_{\beta}^{over}(\mathbf{n})}^{\text{cost of overtainting}} \tag{2}$$

In the next two paragraphs (see B.1 and B.2) we elaborate on the costs $c_{\alpha}^{under}(\mathbf{n})$ and $c_{\beta}^{over}(\mathbf{n})$ and the meaning of α, β. $\tau \in \mathbb{R}^+$ is an input parameter, that dynamically weights the tradeoff between over- and under- tainting. When $\tau = 0$ the cost of overtainting disappears ($0 \cdot c_{\beta}^{over}(\mathbf{n}) = 0$) and,

Table 1. Notation

Notation	Description
$t \in \mathbb{T} = \{t1, t2, \dots\}$	tag type (e.g., network, process, file, etc.)
$i \in \mathbb{N}^+ = \{1, 2, 3, \dots\}$	increasing number that differentiates tags of same type
$\{t, i\}$	unique ID number for each tag
$n_{t,i}$	number of copies of a tag $\{t, i\}$ in memory, i.e. number of bytes that have it
$\mathbf{n}$	2-D optimization vector
α, β	α dictates the fairness in undertainting (tag balancing as $\alpha \to \infty$). β reveals the steepness of the overtainting cost (input par.)
τ	weights the tradeoff between under vs. over tainting (input parameter)
u_t, o_t	importance of different tag types in propagation (u_t) and their contribution to the overtaintingx (o_t) (input parameter)

thus, the undertainting cost dominates, and all tags are propagated. As we increase τ the emphasis moves towards the overtainting, which limits tag propagation. This weighting parameter is often used in *multi-criterion* optimization problems [16, 17]. We are interested in finding *Pareto efficient* operating points [18]. A solution $\mathbf{n}^*$ is Pareto efficient if for any other feasible $\mathbf{n}$

$$c_\alpha^{under}(\mathbf{n}) + \tau \cdot c_\beta^{over}(\mathbf{n}) \le c_\alpha^{under}(\mathbf{n}^*) + \tau \cdot c_\beta^{over}(\mathbf{n}^*)$$
$$\Rightarrow \mathbf{n} = \mathbf{n}^*$$

The above relation suggests that any other solution could improve the undertainting or the overtainting, *but not both.* All Pareto efficient points can be found by scalarization [16], e.g. minimizing $c_{under}(\mathbf{n}) + \tau \cdot c_{over}(\mathbf{n})$ for different values of $\tau \in \mathbb{R}^+$. We will elaborate more on this on Section 6.

(B1. Cost function of undertainting: $c_\alpha^{under}(\mathbf{n})$.) Now, we model the undertainting cost function.

$$c_\alpha^{under}(\mathbf{n}) = \sum_t u_t \sum_i \frac{\left(n_{t,i}\right)^{1-\alpha}}{\alpha - 1}. \quad (3)$$

The fairness parameter α is input and balances the number of propagations for different tags, and u_t weights the importance of different tag types, as explained below. When $\alpha = 1$, the above function is not defined (as $\alpha - 1 \to 0$) and $\log \left(n_{t,i}\right)^{-1}$ is used instead. Note that, the proposed α-fair fairness function was inspired by the fairness in resource allocation for wireless networks [17, 19, 20]. Figure 3(a) depicts this function for different values of α. We now discuss the properties of our considered cost function.

It is *monotonically decreasing* on $n_{i,t}$. This means that the more the copies of a tag, the lower the undertainting cost for that tag. Thus, the slope of undertainting cost is continuously decreasing, meaning that it has negative gradient.

As $\alpha \to \infty$ *tag-balancing is achieved through max-min fairness.* As we increase α the slope becomes more and more steep. As $\alpha \to \infty$ the slope maximizes and thus our function attempts to maximize the propagation of tags with fewer copies, i.e. max-min fairness. The latter maximizes the entropy of the system from an information-theory perspective. This fairness has interesting implications for DIFT systems. For example, assume that a system service reads input from the network that contains the number of bytes that a remote machine will send as an integer, followed by the actual bytes of data. Suppose the service first reads the integer from the network and then allocates enough space on the heap for all the data based on that integer. The entire placement of data on the heap has now been influenced by tagged input. If address dependencies are propagated too aggressively, then for the rest of the lifetime of that process all information stored on the heap will become tainted with the tag that had been associated with that integer (a netflow tag). Other examples include, the scenario where a stack pointer is tainted by variable-sized arrays on the stack, or the stack pointer being popped or set from a register while the program counter happens to be tagged. Then, everything on the stack becomes tainted and starts overtainting all taintable objects in the system because the stack is heavily accessed. Slowinska and Bos [21] provide more examples in that context. In all such scenarios, MITOS will automatically decelerate the propagation of the tags that attempt to hurt the system entropy.

Tag-balancing alone may not be sufficient for a good propagation decision. Different tag types carry heterogeneous information (e.g., network, pointer, file) and potentially propagate differently in the system. This calls for schemes that are able to weight the propagation speed for different tag types, based on e.g. the application, the system workload, or the security policies implemented. Our cost function *flexibly* overcomes this obstacle by using $u_t \in \mathbb{R}^+$ which weights the importance of different tag types and can boost or decelerate their propagation respectively. We define $\mathbf{u}$ to be the vector weighting the different tag types: $\mathbf{u} = [u_{t1}; u_{t2}; \dots]$. This leads to a weighted fair optimization [19].

The cost function $c_\alpha^{under}(\mathbf{n})$ is *convex* for $n \in \mathbb{R}^+$. The function $u_t \cdot \frac{n_{t,i}^{1-\alpha}}{\alpha-1}$ is convex on n since its second derivative is $u_t \cdot \alpha \cdot n^{-\alpha-1} \ge 0$. Finally, the sum of convex functions is also convex [16], and thus $c_\alpha^{under}(\mathbf{n})$ is also convex.

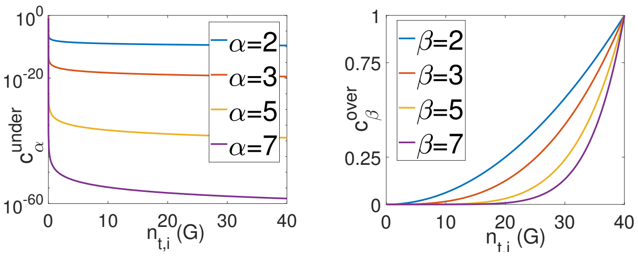
(B2. Cost function of overtainting: $c_{over}(\mathbf{n})$). Overtainting is the phenomenon where an extremely large percentage of bytes carry a large number of tags. This restricts the amount of useful information gained by the DIFT system, and we will refer to it as memory pollution. If R is the memory capacity of the system in bytes (e.g., main memory, register bank, Ethernet card memory) and M_{prov} is the maximum size of the provenance list, then the *total tag space in the provenance lists* is $N_R = R \cdot M_{prov}$. For example, if $R = 4GB$ and for each byte we keep a list up to 10 elements, there are in total $N_R = 40 * 10^9$ provenance list elements. Then,

$$c_{\beta}^{over}(\mathbf{n}) = \left(\frac{\sum_t o_t \sum_i n_{t,i}}{N_R}\right)^{\beta} \tag{4}$$

Parameter β is an input and dictates the slope, namely steepness, on the overtainting cost. Figure 3(b) depicts the function of Eq. (4). This cost function has the following properties.

It is *monotonically increasing* on $n_{i,t}$, i.e. the larger the number of tags in the system, the higher the cost. Thus, its slope is continuously increasing with positive gradient. Following the standard penalty functions, it should have at least quadratic penalty on the memory pollution, thus we keep $\beta \geq 2$, ensuring also that it is twice differentiable [16]. As β increases the cost of overtainting gets steeper.

Similarly to the undertainting cost, different tag types may impact memory pollution differently. Our cost function *flexibly* takes memory pollution into account by using $\mathbf{o} = [o_{t1}; o_{t2}; ...]$ that weights the partial pollution of different tag types and adapts their impact on the total pollution. Finally, the function $c_{\alpha}^{over}(\mathbf{n})$ is *convex* for $n \in \mathbb{R}^+$ [16].



(a) undertainting cost c_{α}^{under}. (b) overtainting cost c_{β}^{over}.

Figure 3. Considered cost functions for under/over tainting.

(C. Optimization Problem). Based on the defined control variables and cost function we formulate our problem.

Problem 1. *The problem formulation for the indirect flow (IF) dilemma is:*

$$\min_{\mathbf{n}} . \sum_t u_t \sum_i \frac{(n_{t,i})^{1-\alpha}}{\alpha - 1} + \tau \cdot \left(\frac{\sum\limits_t o_t \sum\limits_i n_{t,i}}{R}\right)^{\beta} \tag{5}$$

$$N_R - \sum_t \sum_i n_{i,j} \geq 0 \tag{6}$$

$$N_R - n_{t,j} \geq 0, \forall t \in \mathbb{T}, i \in \mathbb{N}^+ \tag{7}$$

Our control variable is the vector $\mathbf{n}$ that determines the decision of propagating the tags coming from indirect flows. Specifically, if a tag in that case is worth propagating we further increase the corresponding value of $\mathbf{n}$ (see A. Control Variables). The cost function that we intent to minimize, at Eq. (5) is the weighted tradeoff between under- and overtainting coupled with the other properties and variables (see B. Cost Function). Constraint Eq.(6) is linear and states that the total number of tag copies should not exceed the total tag space in the provenance lists. Constraint Eq.(7) ensures that each tag should not have more copies than the total number of memory bytes (i.e., no memory byte is allowed to have more than one copy of a certain tag).

This problem has many challenges. First, the control variables $n_{t,i}$ of the considered vector $\mathbf{n}$ can only take integer values, i.e. $n_{t,i} = 1, 2, 3, ...$, since a tag can be propagated to one or several bytes. Thus, even though the cost function is convex, the feasibility region transforms it into an *integer optimization problem*, making the problem NP-hard, which is hard to solve optimally. Second, the number of control variables $n_{t,i}$ can experience sharp increases and decreases in a very short interval (e.g. a video game reads data from files and downloads content from Internet, thus generates hundreds of file, network tags in a few mseconds), by continuously changing the dimensions of $\mathbf{n}$. This further complicates the problem since the system dynamics and the optimal points change continuously as the system runs.

4.2 Solution: Distributed Optimization Algorithm and Policies

We now tackle Problem 1. We start by discussing how we are going address the two major challenges discussed earlier (the problem is NP hard and the dimensions of the control variable change continuously). We first propose to *relax the allowed values for* $n_{t,i} \in \mathbb{N}$, and assume that $n_{t,i} \in \mathbb{R}^+$. The continuous relaxation of the problem leads to a convex optimization problem which can be solved analytically using the method of Lagrange multipliers and Karush Kuhn Tucker (KKT) conditions [16], to derive the optimal vector $\mathbf{n}^*$. [1]

However, such a solution would require a centralized implementation that would need to (i) gather all the necessary information from all tags, and (ii) re-calculate the global optimal point every time a new tag is inserted/deleted at the system. Note that, this solution might not scale well, as new tags are created and deleted very frequently as the system runs, and, thus, the overhead in re-calculating the new optimal points can be prohibitive. Thus, we propose a *distributed solution* that scales well, namely, it dynamically adapts to the rapid tag creations/deletions and the system changes, and show that it converges to an optimal point in the long term.

Solution roadmap: In the following, we start with Indirect Flow Propagation IFP Scenario 1, where we assume that the source operand of the indirect flow attempts to propagate a single tag and the destination has at least one available space in its provenance list to accommodate it. Then, we generalize it to IFP Scenario 2 and we assume that the destination has

[1]Note that in practice, one could round these values to the closest integer to get an approximately optimal solution.

limited available space in its provenance list, and cannot accommodate all tags scheduled for propagation.

IFP Scenario 1: Single tag propagation with sufficient space at the destination provenance list.

Assume an indirect flow scenario in a particular instruction where (i) the source operand has only one tag for potential propagation. Also, (ii) the destination has (at least) one available space in its provenance list, e.g., see Fig. 4. The store word instruction in Fig. 4 copies data from a register to memory. In our example, it attempts to store a word from register t_0 to the memory location corresponding to $7FFFF0 + t_3 = 7FFFF0 + 8 = 7FFFF8$, given that the value of $t_3 = 8$. This is an address dependency, since the value of t_3 will dictate the memory address location that the data of register t_0 will be stored, and further the system execution. Note that, there is a direct flow too from t_0 to $7FFFF8$ that will be propagated following the basic DIFT rules (see Section 2) and is out of the scope of this paper.

Our objective is to answer the following question: *should the DIFT system propagate the red tag C?* Algorithm 1 shows our proposed method towards answering this question. The main idea is to take the indirect flow propagation decision based on the first-order optimization criterion [16].

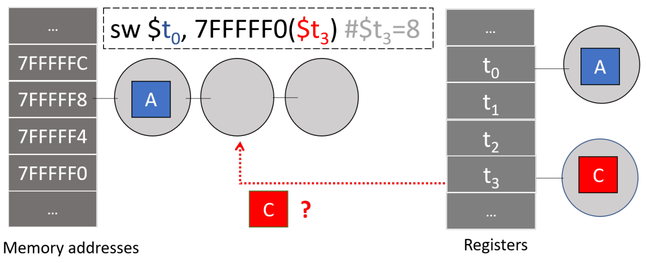


Figure 4. The store word *sw* instruction copies data from the register t_0 to memory location $7FFFF0+t_3 = 7FFFF0+8 = 7FFFF8$. This involves an address dependency, since the value of t_3 dictates the memory address that the data of register t_0 will be stored. Thus, we need to track the flow from t_3 to memory location $7FFFF8$: note that the list of $7FFFF8$ has at least one space to accommodate the single tag potentially coming from t_3. Alg. 1 answers the question: should the red tag C be propagated from t_3 to $7FFFF8$?

Algorithm 1 IFP Scenario 1: Single tag propagation with ID $\{T, I\}$ with sufficient space in the provenance list of the destination.

```
1: Step1: Derive the direction of the gradient towards $n_{T,I}$.
2:    $\Delta n_{T,I} = \frac{\partial}{\partial n_{T,I}} c(\mathbf{n})$ with Eq. (8).
3: Step2: Using the gradient-descent criterion decide about
   the IFP.
4:    If $\Delta(n_{T,I}) \leq 0$ then propagate the tag $\{T, I\}$.
5:    Else, do not propagate the tag $\{T, I\}$.
```

In the following two paragraphs we elaborate on the two-steps of Algorithm 1.

First, we derive the direction of the gradient towards the dimension we are interested in. In our case, this dimension refers to the control variable that is associated with the tag considered for indirect flow propagation. For the sake of presentation, we assume that the type of the tag considered for propagation (e.g., the red tag in Fig. 4) is T, with identification number I, i.e. the involved control variable $n_{T,I}$. The partial derivative of the tag involved in an indirect flow with ID $n_{T,I}$, namely $\Delta n_{T,I}$, that will determine its propagation is:

$$\begin{aligned}\Delta n_{T,I} &= \frac{\partial}{\partial n_{T,I}} \cdot c(\mathbf{n}) = \\ &= -u_t \cdot \left(n_{T,I}\right)^{-\alpha} + \tau \cdot \beta \cdot \left(\frac{\sum_t o_t \sum_i n_{T,I}}{N_R}\right)^{\beta-1}\end{aligned} \tag{8}$$

This quantity refers to the cost added by propagating that tag to one more byte, and, therefore, can be seen as the *marginal cost* of the indirect flow propagation. This marginal cost depends on: (i) the submarginal cost of undertainting that attempts to decrease it ($-\alpha \cdot u_t \leq 0$), and on (ii) the submarginal cost of overtainting that attempts to increase it ($\beta \cdot \left(\frac{\sum_t o_t \sum_i n_{T,I}}{N_R}\right)^{\beta-1} \geq 0$). The sign of their sum, and, thus, the direction of the gradient, follows the sign of the highest absolute value.

Second, following the direction of the gradient, we attempt to improve our considered cost function. Since our function is convex and we attempt to minimize it, we need to follow the opposite direction of the considered gradient [16]. More precisely, if the partial derivative of Eq. (8) is negative, then the first-order optimization criterion suggests to increase the involved control variable by +1 and thus to propagate the indirect flow. On the other hand, if the partial derivative is positive such a decision would hurt our objective and thus the DIFT system should not propagate the tag.

Lemma 4.1. ***[Optimal decisioning for indirect flow propagation]*** *The optimal rule for determining the propagation of a tag currently involved in an indirect flow, is:*

$$\textit{propagate it if: } \Delta n_{T,I} \leq 0, \textit{ block it otherwise.} \tag{9}$$

IFP Scenario 2: Multiple tag propagations with insufficient space at the destination provenance list.

We now generalize the above scenario. Assume an indirect flow scenario where (i) the source operand has multiple tags for potential propagation, and (ii) the destination operand does not have enough space in its provenance list to accommodate all the potential tags scheduled for propagation (e.g., see Fig. 5). In this store word instruction we see again an address dependency from the register t_3 (source) to the same memory location 7FFFFF8 (destination). However, now the source has three potential tags for propagation and the destination only two available space positions in its provenance list. Our objective is to answer the following question: *which, at maximum two, tags out of the C, E, B should the DIFT system propagate ?* This is a challenging question and considering all possible combinations would require exponential complexity. We extend Algorithm 1 to Algorithm 2, and we use a prioritized first-order optimization criterion, by exploiting the fact that the improvement of the cost function can be maximized if we follow the gradient towards the lowest submarginal costs.

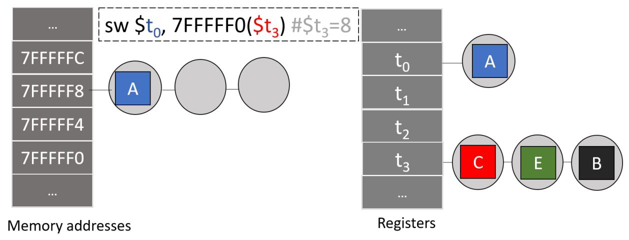


Figure 5. The *sw* attempts to copy data from the register t_0 to memory location $7FFFFF0 + t_3 = 7FFFFF0 + 8 = 7FFFFF8$, that does not have enough space: the list of $7FFFFF8$ has available space for two tags, and the list of t_3 has three tags. Alg. 2 answers the question: which, at maximum two tags, should be propagated from t_3 to $7FFFFF8$?

Algorithm 2 IFP Scenario 2: Propagation of multiple tags to a byte with limited available space in its provenance list, namely A available tags space.

```
1:  Derive the partial derivatives (marginal costs) for all
    involved tags using Eq. (8).
2:  Sort the tags wrt their partial derivatives increasingly:
    $\{\Delta n_{T_1,I_1}, \Delta n_{T_2,I_2}, \dots\}$, such that $\Delta n_{T_1,I_1} \leq \Delta n_{T_2,I_2} \leq \dots$.
3:  Set $j = 1$. // tag being considered currently for propagation.
4:  Set #props= 0. // number of successfully propagated tags.
5:  while (#props $\leq$ A) and ($\Delta(n_{T_j,I_j}) \leq 0$)
6:      Propagate tag $j$.
7:      #props++. // increase by 1 the propagated tags.
8:      j++ //move for the next tag.
9:      Recalculate $\Delta(n_{T_j,I_j})$ of the tag $j$.
10: end.
```

First, we derive all the partial derivatives of all tags involved in the considered indirect flow at the source using Lemma 4.1. (line 1, Alg. 2). Then, we sort the partial derivatives in an increasing order, such that the first tag (j=1) has the lowest marginal cost (line 2, Alg. 2). [2] Then, we set (i) the first tag to be considered for propagation to the one with lowest marginal cost i.e. j=1, and (ii) the tags that have been successfully propagated to 0, *#props* = 0 (line 3-4, Alg. 2). The while loop that follows keeps propagating the tags while the available space in the destination provenance list is not exceeded (#props $\leq$ A) and while the marginal cost of the current tag is negative ($\Delta(n_{T_j,I_j}) \leq 0$). More precisely, if the above two conditions hold true, we propagate the tag and increase by +1 the counter of propagations (line 6-7, Alg. 2). Then, we move the pointer to the next tag (line 8, Alg. 2) and recalculate the partial derivative of the next tag since the cost of overtainting might have changed (line 9, Alg. 2).

Since the tags are ranked according to their marginal cost, and the propagation decision is based on them, we claim that during each indirect flow the improvement in $c(\mathbf{n})$ will be maximal among all feasible directions (a variable $n_{t,i}$ cannot change during an indirect flow propagation, if the tag $\{T, I\}$ is not present in the provenance list of the source); given the convexity of our cost function, this method is shown to correspond to a distributed implementation of a gradient decent algorithm [22].

We now discuss the properties of our proposed rule in Lemma 4.1 and of our generic Algorithm 2.

- Our rule for the IFP is of *low-complexity*. Specifically, the time complexity is $O(1)$, since every time MITOS needs to make an IFP decision it only needs to sum two integers and then find the sum's sign (see Alg. 2). Regarding the space complexity, we need (i) $O(N_R)$ space for the submarginal cost of undertainting (left part of Eq. (8)), as our policy is byte-level attributable. Also, we only require (ii) $O(1)$ space for the overtainting cost, as we keep a single estimation of the memory pollution (right part of Eq. (8)).
- It is *distributed*: upon an IFP of a certain tag, MITOS only needs to retrieve a (local) value about how undertainted the tag is, and an (global) estimation of memory pollution (see e.g. Eq. 8). This makes MITOS highly *scalable* since the complexity does not change regardless the number of tags and the rapid changes on the control vector $\mathbf{n}$.
- It is *flexible*, since by changing the various input parameters one can flexibly weight the involved tradeoffs differently. It is also *α-fair*, since α captures different degrees of tag balancing.
- Our proposed algorithm provably converges to an optimal point optimizing the involved tradeoffs.

4.3 Generalization to direct flows and different objectives.

Modern instruction-level DIFT systems usually incur an intolerable performance overhead due to aggressive direct flow propagations [12]. MITOS can be extended to optimize the propagation of direct flows theough the considered cost function. In that case, Alg. 2 should be invoked every time the system has to make either direct or indirect flow decision.

[2]Note that the overtainting cost is the same for all, so the cost of undertainting will affect the order at this point.

Additionally, MITOS can be used to optimize different cost functions for propagation decisioning. One should just: (i) change the cost function by modeling the tradeoffs he wants (for more details see Section 7), (ii) use the new partial derivative in the tag sorting (line 2, Alg. 2) and then check whether the partial derivative is negative to perform the tag propagation (line 5, Alg. 2). The nature of our proposed algorithm would stay the same (e.g., it will still be scalable, of low complexity, convergent, etc.).

5 Implementation On an Existing DIFT System

In this section we detail how we implemented MITOS into: FAROS, a software-based DIFT system that is publicly available [12, 23]. As seen in Fig. 6, our architecture consists of five different layers. From a bottom-up viewpoint, we note: the host that is a Linux 14.04 machine (orange color), PANDA, FAROS, MITOS (green color), and finally (iii) Windows 7 as guest machine (blue color). We have implemented all functionalities and interactions associated with MITOS.

PANDA. PANDA is built upon the QEMU whole-system emulator. It adds to QEMU various features for instruction level analysis including the capability of: (i) *recording* a run, and later (ii) *replaying* it (multiple times if needed) enabling instruction-emulation and analysis. This capability of multiple replays is suitable for our analyses as we want to consider the impact of different inputs into the system.

FAROS. FAROS is a DIFT-based reverse engineering tool with direct flow propagations to be applied mostly for malware analysis. FAROS supports four types of tags: (i) *process or cr3*, which is added to all bytes touched by a process, (ii) *netflow* tag, which is added to every byte of a packet coming from network, (iii) *file*, which is added to file bytes when loaded into the memory, and (iv) *export-table*, for bytes belonging to the memory area where Windows performs its linking and loading operations. FAROS stores tags in separate hashmaps in the host machine memory.

We now explain why we chose FAROS to evaluate MITOS. FAROS suggest that the dilemma of indirect flow propagation is overcame via the application of tags of different types in a per-security fashion. Specifically, they propose to regain the information flow lost by not tracking indirect flows by applying tags with different meanings and observe their behavior according to the goals of a specific security policy. For example, they showed that when one export-table tag and one netflow tag come together on a byte, they could flag in-memory-only attacks. However, this direct flow-based heuristic does not combat the indirect flow dilemma or the tradeoffs encountered there. We go a step beyond and we suggest that, instead of recouping the information flow lost from not propagating indirect flows with tag differentiation, one can be leverage the latter towards the indirect flow decisioning along with our analysis for other tradeoffs. Finally, FAROS has prohibitive performance overhead for on-the-fly application, and it presents poor detection efficiency for in-memory attacks.

We now describe the implementation of MITOS along with its interaction with PANDA and FAROS in Fig. 6. PANDA provides access to all instructions emulated in a previously recorded run (steps (1)-(2)). Then, FAROS component *is_DFP* filters and processes the instructions that involve a direct flow propagation (DFP) (step (3)), and propagates *all* the DFPs by inspecting and modifying the shadow memory at the host. Then, FAROS gives the token to MITOS, to propagate any indirect flow propagations (IFPs). We have designed the module *is_IFP* to filter and process the instructions that carry IFP, i.e., address or control dependencies (step (4)). Address dependencies happen in instructions like store/ load word (e.g., *sw*, *lw*), and control dependencies happen in instructions referring to branches (e.g., *br*, *beq*). Next, the instructions associated with IFP are subject to Alg. 2 (step (5)). Specifically, by inspecting the shadow memory, MITOS calculates the marginal costs $\Delta n_{T,I}$ for all tags T, I appeared in the source of the instruction, it sorts them and decides if they worth being propagated (see Alg. 2). Finally, MITOS updates the shadow memory for the tags that got propagated.

MITOS can also track DFP, which we investigate in a study case in Section 6.2. For this case, the functions *is_DFP* and *DFP* are removed from FAROS. Then, both direct and indirect flows are forwarded to MITOS and further to Alg. 2. To do so, it suffices to replace the function *is_IFP* with *is_DFP_or_IFP*, i.e. MITOS processes not only the instructions associated with address/control dependencies (IFP), but also the ones related to copy, union and deletions instructions (DFP) (e.g., and, move, or, xori, etc.).

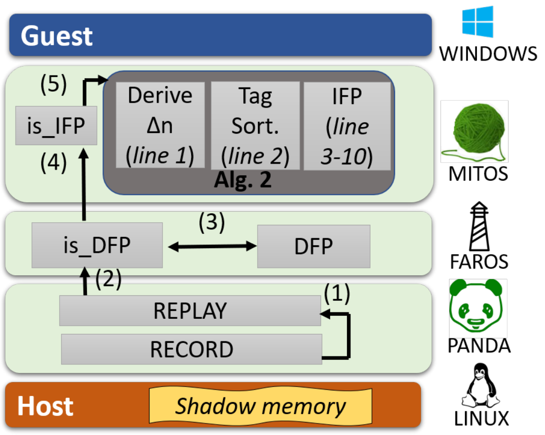


Figure 6. MITOS implementation.

6 Performance Evaluation

Here we evaluate MITOS' performance under various tradeoffs encountered in different indirect flow scenarios, such as undertainting vs. overtainting, tag type importance, and different fairness degrees in tag balancing (Section 6.1). Then, we evaluate MITOS in a study case, where FAROS is flagging in-memory-only attacks (Section 6.2). We show how the application of MITOS for all types of flows can not only improve FAROS' spatiotemporal performance, but also its detection accuracy, in terms of recognizing the bytes that are part of an exploit. In the following, if not explicitly mentioned, we assume that $\alpha = 1.5$, $\beta = 2$, $u_t = o_t = 1\ \forall t \in \mathbb{T}$, $\tau = 1$, and that all τ values are normalized up to the power of 10^6. We have varied the parameters in our evaluations and have reached similar conclusions. All experiments were conducted on a system with an Intel Core i7-7700K 2.90GHz processor, and 32G RAM running Linux Ubuntu 14.04. The guest was Windows 7 Ultimate 32-bit with 4GB memory.

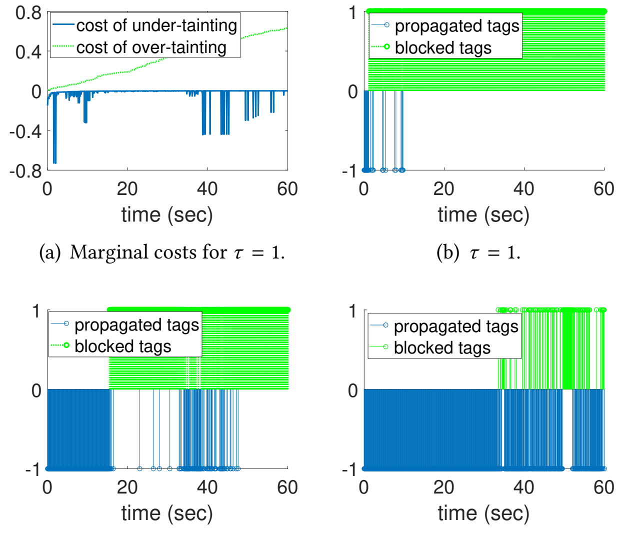
6.1 Sensitivity Analysis

Sensitivity analysis explores the impact of the inputs in a mathematical model (e.g., MITOS inputs include τ, α, u_t) to the output (e.g., in MITOS we are interesting in the indirect flow decision impact, memory pollution, overhead etc). In this section we focus on three different scenarios and investigate MITOS' performance on the tradeoffs involved in the indirect flow decision-making.

(Scenario 1: Network (1 minute record - 11 hours of replay)) We ran an one-minute network-benchmark on Windows using the PerformanceTest tool of Passmark [24], where the guest acts as a client and downloads several MBytes of data from a remote server. We replayed this recording multiple times with MITOS on top of FAROS using different values for the input parameters (please see Table 1 for the input parameters). Below, we focus on the impact of our inputs on the tradeoffs involved in indirect flow decisioning.

Under vs. overtainting. The parameter that weights this tradeoff is $\tau > \mathbb{R}^+$, where the higher the τ the more emphasis put on the cost of overtainting and the less emphasis is out on indirect flow propagations. Fig. 7 shows how the system reacts for three different values of τ. We replay the one-minute recordings three times, using different values of $\tau = 1, 10^{-1}, 10^{-2}$, keeping all the other parameters fixed, and wait until the system converges to a point at the end of the replay (actually, the control vector $\mathbf{n}$ converges to a value). Fig. 7(a) shows the marginal costs of under- and over- tainting for different indirect flow propagations that MITOS encountered as a function of time following Eq. (8). For the sake of presentation, we have included the effect of τ at the cost of overtainting. The undertainting costs of different indirect flows varies; this cost is only dependent on the current number of copies of the considered tag. The overtainting cost is (mostly) monotonically increasing on time since the memory pollution is (mostly) increasing on time due to the new tag insertions/propagations. Figure 7(b) shows the corresponding decisions for these indirect flows: if the cost of undertainting dominates the tag is propagated (and we plot +1 in the y-axis), otherwise the tag is blocked (and plot -1). Since we keep a relatively high value of τ, most of the tags are blocked. In Fig. 7(c), 7(d) we decrease τ, which further decreases the emphasis of overtainting and plot the decisions of the indirect flows encountered. Indeed, more tags get propagated over time due to the τ decrease.

Message: The tradeoff between undertainting and overtainting should not be a challenge in the indirect flow dilemma and further in the DIFT efficiency. Instead of applying simple heuristics that attempt to improve either side of that tradeoff, MITOS weights it with respect to τ and converges to a point that *flexibly* optimizes the tradeoff for that τ.



(a) Marginal costs for $\tau = 1$. (b) $\tau = 1$.

(c) IFP decisions for $\tau = 0.1$. (d) IFP decisions for $\tau = 0.01$.

Figure 7. τ vs. under/overtainting marg. costs, IFP decisions.

Pareto optimal distribution. We showed that for a fixed value of τ, MITOS converges to an optimal point. This point is a Pareto optimal point, as any other solution could perhaps improve either cost *but not both.* [3] Fig. 8 depicts all Pareto optimal points for different values of τ. Each point in the x-axis (different τ value) corresponds to a MITOS' run and the y-axis ($c_{\alpha,\beta}(n)$) corresponds to the value of the cost function.

Message: The pareto distribution is quite useful when one wants to investigate, or even predict, how a change in τ affects the cost function and performance.

Fairness - Tag balancing. The higher the α, the more fair MITOS is (see Eq. (3)). Specifically, as $\alpha \longrightarrow \infty$ the undertainting cost becomes steeper, i.e. it penalizes more intensely the overpropagated tags. The latter attempts to maximize the tags with fewer copies. In other words, through a max-min fairness MITOS can perform *tag balancing* based on the α

[3]It's possible that some propagations that would considerably improve the cost function might have been blocked due to an earlier suboptimal decision. However, in the long-term that this phenomenon averages out [22].

input. Fig. 9 corresponds to six different MITOS' runs for six different values of α. We measure the fairness degree, or taint-balancing efficiency, based on the mean square error difference between the number of copies of different tags. The sharp deviations of the tags can be alleviated by adapting α, thus improving tag balancing performance up to 2×. This is important as traditional DIFT systems tend to over-propagate tags in multiple scenarios, consequently hurting their overall performance and wasting memory resources from the provenance lists [12].

Message: MITOS input parameter α flexibly captures different fairness degrees, in terms of tag-balancing. We envision this to have immense impact on modern DIFT systems, since they experience situations that tend to over-propagate certain tags. MITOS is generic enough to capture all these cases and handle tag balancing an optimal way.

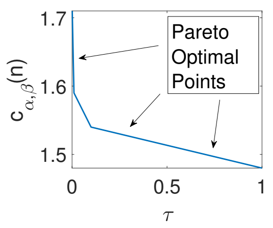


Figure 8. τ vs. Pareto optimal points (network ben.).

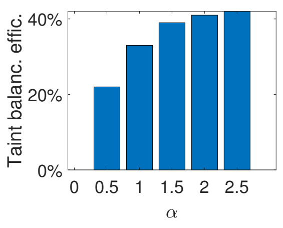


Figure 9. α vs. fairness and tag-balancing.

Tag type importance. We now focus on the tradeoff arising when tags of different tag types compete for propagation. Modern DIFT systems might include different tag types with different properties, importance, and , propagation speeds [12]. There are many reasons that warrant dynamic and somewhat personalized strategies to determine the weight of over vs under tainting based on tag type. For example, for applications handling sensitive files might want to put a higher cost on undertainting so that the probability of not tracking a file tag is low. MITOS takes this into account through the parameters u_t (determining the importance of a particular tag type), for all tag types $\tau \in \mathbb{T}$. In Fig. 10 we consider different values of u_{netflow} (by keeping the remainder parameters fixed and equal to 1). For each value we plot in blue (netflow line in the legend) the proportion of netflow tags, encountered at the end of each replay, that is normalized by the maximum value taken when $u_{\text{netflow}} = 100$. Increasing u_{netflow} monotonically boosts the netflow tag propagation speed. The boosting of certain tag type propagation speed impacts how MITOS handles other tag types, because a speed boosting means an increase in memory pollution. For example, as shown in Fig 9(a) the number of propagation of CR3 tags decrease as MITOS increases the important of netflow tags. Note that, since export table tags are, in general, more mildly propagated compared to the CR3 tags, the undertainting cost of the former is higher, and, thus, their propagation speed is mildly decelerated. Results for file tags are not shown because there were no indirect flow propagations for them for this particular benchmark.

Message: MITOS introduces a flexible way to dynamically fine-tune the propagation speed of different tag types through u_t, to accommodate the inherently heterogeneity of tag priorities for a particular system.

(Scenario 2,3: CPU and file-system (1 minute record - 12 and 11 hours of replay)) We evaluated MITOS in two additional scenarios: leveraging (i) a CPU-benchmark using the PerformanceTest tool of Passmark, where the guest OS creates, runs and terminates various processes performing tasks related to compression, physics, and prime numbers and (ii) a file-benchmark, by manually creating, copying and pasting various files including other actions that would impact indirect flows e.g. search and replace. We could not use the file-benchmark of Passmark because it only uses basic file actions, which would not lead to indirect flows. For these two scenarios we found results similar to those detailed for Scenario 1 (Network). For completeness we plot the pareto optimal distributions in Fig. 11. The File benchmark creates some indirect flows, with few tags. Thus, increasing τ will not considerably affect the cost function after some point. The CPU benchmark is steeper compared to both the Network and File benchmarks, since CR3 tags are significantly more involved in indirect flows.

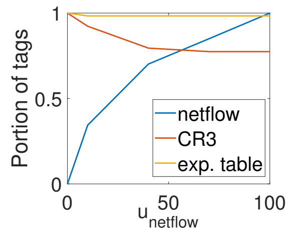


Figure 10. u_{netflow} vs. portion of netflow tag propag.

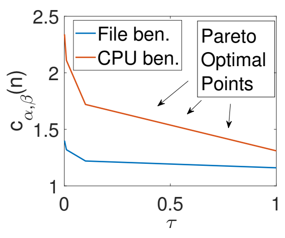


Figure 11. τ vs. Pareto optimal points (file, CPU ben.).

6.2 Case study: Flagging In-Memory-Only Attacks

In the following we apply MITOS in FAROS while it is flagging sophisticated *in-memory-only attacks* and show the substantial improvement MITOS brings in spatiotemporal performance and detection efficiency. In particular, we study how much time MITOS/FAROS needs to replay such an attack compared to the standard FAROS (time complexity), how much memory is used (space complexity), and how many bytes it successfully detected as suspicious (detection efficiency). To do so, we generalize MITOS to also capture and optimize direct flows, as explained in Sections 4.3, 5.

In an in-memory-only attack, the attacker, usually through a shell, injects a payload inside a legitimate process address space. The hallmark of the in-memory-only attack is the following. The payload bytes come from the Internet and, thus, are associated with the tag *netflow*. Then, these bytes are written into the kernel memory area where linking/loading operations occurs and are also associated with the tag *export-table*. FAROS flags the attack when these two tags (netflow and export-table) come together on a byte.

We implemented the in-memory attacks using the Meterpreter module from Metasploit in a way similar to that done for FAROS [12]. We set up the attacker's virtual machine (Linux Kali) and generated a shell code that runs in the victim's virtual machine (Windows 7). This opens a session for the attacker and we then perform a remote reflective DLL injection targeting the *calculator.exe* from the attacker. In our evaluations we leveraged two replays using the same record: (i) propagating all direct flows and no indirect flowssuch as done in FAROS and (ii) with MITOS and propagating all flows (direct and indirect) improve the considered cost function (see Alg. 2). We used the following input parameters: $u_{\text{netflow}} = u_{expt} = 1$ to boost the propagation of the tags we are interested in. We set $u_{CR3} = 0.01$ because in Windows the CR3 tags propagate quite aggressively in the address process space, and thus need to be decelerated. We also set the fairness parameter to $\alpha = 3$ to mildly stop any potential tags that overpropagate.

The spatiotemporal complexity and the detection performance of both systems is depicted in Table 2. We ran six Metasploit shells (reverse https, reverse https proxy, reverse tcp rc4 dns, reverse tcp rc4, reverse tcp) and show the average performance. The time and space needed to replay the recorded attack improves $1.65X$ and $1.11X$ times, respectively, when MITOS is used to make propagation decisions. MITOS propagates only the tags that are important information flow-wise. Even though MITOS propagates fewer tags, thus alleviating the CPU and memory overheads, it is able to successfully detect 2.67× times more bytes that were involved in the in-memory attack. This happens since: (i) $\alpha = 3$ improves tag balancing, leaving more space for the tags carrying more important information, (ii) $u_{CR3} = 0.01$ decelerates the propagation of CR3 tags that occupy a substantial part of the provenance lists.

Message: MITOS propagates only the tags that improve the considered cost function. As this minimizes unnecessary propagations, the spatiotemporal complexity is improved: the provenance lists carry fewer elements and, thus, their traversing time decrease. Also, as the detection efficiency of DIFT heavily depends on the provenance list management, MITOS has an immense improvement on detecting the bytes involved in an in-memory-only attack.

	FAROS	**MITOS**	**Improvement**
Time (sec)	837	509	1.65×
Space (Mbytes)	2.21	1.99	1.11×
Detected bytes	543	1449	2.67 ×

Table 2. Time, space complexity and the number of bytes that have successfully being detected of being involved in an in-memory attack for: FAROS and FAROS with MITOS.

7 Discussion and Future work

In summary, MITOS consists of an analytical framework and policies the for indirect flow propagation in DIFT systems. From a theoretical viewpoint, MITOS is novel as it (i) *analytically* studies the tradeoff between undertainting and overtatinting, and between the importance of different tag types. It also (ii) introduces *fairness* and *tag-balancing*: key properties for the success of DIFT. While the complete problem is shown to be NP-hard, MITOS (iii) provides the optimal rules for indirect flow propagation that converge to an optimal point *flexibly* optimizing performance with respect to the input parameters. From a systems viewpoint, MITOS is distributed, scalable, of low complexity and thus applicable to most DIFT systems, and suggests that indirect flows should not impede the widespread application of DIFT; instead the DIFT system should propagate them following the application that improve the knowledge of information flow. When implemented in a real software-based DIFT system (FAROS[12]), our sensitivity analysis provided several insights about the involved tradeoffs (see *Messages* in Section 6.1). Our study case with in-memory-only attacks showed that MITOS improved FAROS in the following respects: spatiotemporal complexity improved up to 1.65× (amount of time and memory needed for the taint analysis) and the detection efficiency (how many bytes got successfully detected as suspicious) up to 2.67×.

We now discuss some limitations from the theory viewpoint, and future work plans. Parameter τ weights the tradeoff between under- and over- tainting. While one can easily define a decent value of τ and keep it fixed, we believe that τ should ideally change throughout the program execution to boost performance (e.g., based on the belief of the system if it's under attack, or the current memory pollution). Considering τ as an additional control variable, complicates the problem and we defer to future work its optimal solution. Additionally, we have assumed that the provenance lists follow a First-In-First-Out (FIFO) queue: we drop the head of the list if the list is full and an additional tag attempts to enter. We defer to future work the design of a proper tag scheduling and dropping decisioning using penalty functions for indirect flows, as Matzakos et al. did for delay tolerant networks [22]. We also plan to modify our cost-function to consider additional context for the tag propagation decisioning (see Section 4.3). For example, how often tags are propagated at specific program counter locations, the types of the data being processed e.g., character from a string vs. entropy from the entropy pool, or the confluence of different types of data at a program counter location.

Note that, as our implementation was based on FAROS and PANDA, we encountered several limitations in the performance evaluation. For example, FAROS poses a large overhead on the host machine: e.g., the memory required to replay a record increases exponentially on the record duration, prohibiting us to run scenarios longer of one minute. Also, PANDA restricts the size of the record and further the system activities that can be recorded simultaneously. The latter prevented us from running complex evaluation scenarios, e.g., run multiple attacks of benchmark scenarios jointly.

We now discuss how MITOS can be implemented in hardware. To ensure implementation flexibility for different hardware platforms, MITOS can be implemented as a configurable component in a System on Chip (SoC). Configuration parameters for the MITOS algorithm can be saved in newly added model specific registers, allowing an interface to a trusted operating system module or platform loader to set up the interfaces. Information flow during execution, tag information can be stored in dictionary-like structures that reside in a segmented portion of main memory. Segmentation can be performed during platform initialization, such as the Pre-EFI Initialization (PEI) portion of Unified Extensible Firmware Interface (UEFI), much like the enclave page cache is reserved for usage in Intel's Software Guard Extensions (SGX). Recently accessed information can be stored in a MITOS-specialized series of caches to mask memory latency. We move the computational process employed by MITOS to decide tag propagation to specialized hardware. We extract data flow information directly from the CPU as code executes. For out of order cores, we look at the commit stage in the CPU, as to capture the proper architectural state and not violate execution model. The decision on whether to propagate tag information is then performed by hardware. Because the segmented portion of memory is limited in size, it may need to be swapped. We can perform this action by relying on the operating system to swap the information for us, in which case it must be stored encrypted and cryptographically signed, or through trusted service into a trusted storage area.

8 Related Work

DIFT in the context of detecting attacks was co-introduced by Costa et al. and Suh et al. [1, 6]. TaintBochs [25] was another early application of taint analysis to analyze data lifetime in a full system. Other early DIFT works include [14, 26, 27] that explore various policy tradeoffs for DIFT schemes and higher-level systems issues, vulnerabilities in commodity software and honeypot technologies. Malware analysis [7, 28], network protocol analysis [5, 29] and datataflow tomography [12, 30], full-system recovery after memory corruption attacks and protecting kernel integrity against rootkits [31, 32] are other directions that DIFT is leveraged.

Most past work on DIFT focused on software implementation and did not satisfactory address indirect flows. The earliest DIFT papers identified the problems with address and control dependencies include [6, 14]. There, it's claimed that (i) address dependencies are propagated for 8- and 16-bit loads and stores, and blocked for 32-bit loads and stores. Additionally, in order to mitigate control dependencies they suggest that (ii) 8- and 16-bit immediate values (i.e., constants that are compiled into the program's machine code) were tainted automatically even if the code did not come from a tainted source. Thus, the authors were able to detect several attacks. More details and analysis of these issues followed [33–36], including a quantitative analysis of full-system pointer tainting [21]. More recent DIFT systems that are designed for flexibility [37, 38] enable address and/or control dependencies to be tracked, but provide no satisfactory method for doing so in practice. Panorama [7] relies on a human to manually label which address and control dependencies tags should be propagated. DTA++ [8] or DYTAN [9] rely on off-line analysis, which does not scale to full systems. Systems designed with correctness as the primary goal, such as RIFLE [10], and GLIFT [11], propagate all tags all the time unless a compiler statically analyzes the information flow and deems a particular operation to be safe.

Some recent schemes that attempt to address include FAROS [12] and V-DIFT [2]. In the former, the authors attempt to regain the information flow lost when indirect flows are not propagated using tag differentiation. In the latter, the authors using linear algebra attempt to approximate a quantitative information flow for indirect flows in an offline analysis. An example of DIFT in mobile communications is TaintDroid [3]. There, the authors detect data leakage of Android applications using variable level tracking within the virtual machine. It works by applying a heuristic that propagates tags from several inputs to that of the return value of functions. While there are some attempts at implementing DIFT in hardware [7, 14, 35, 39–47], they do not handle indirect flows and taint analysis performance overheads - they re-engineer certain hardware components to boost the performance, are developed for hardware emulators, or are limited in the flows of information that can be captured.

9 Conclusion

In this paper we propose MITOS, a framework and set of policies for optimal propagation decisioning under an indirect flow. We have taken into consideration multiple tag types, the undertainting versus overtainting tradeoff and different α-fairness degrees and explain how the latter relates to taint balancing. Our analytical work opens new ways of how DIFT systems should measure and optimize their performance. Experimental evaluation shed light on the complex problem of indirect flow propagation, and shows how performance can be improved up to 167% times under certain scenarios.

References

[1] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-end containment of internet worms," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, 2005.

[2] A. M. Espinoza, J. Knockel, P. Comesaña-Alfaro, and J. R. Crandall, "V-dift: Vector-based dynamic information flow tracking with application to locating cryptographic keys for reverse engineering," in *IEEE International Conference on Availability, Reliability and Security (ARES)*, 2016.

[3] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, 2014.

[4] J. Shin, H. Zhang, J. Lee, I. Heo, Y.-Y. Chen, R. Lee, and Y. Paek, "A hardware-based technique for efficient implicit information flow tracking," in *IEEE Computer-Aided Design (ICCAD)*, 2016.

[5] G. Wondracek, P. M. Comparetti, C. Kruegel, E. Kirda, and S. S. S. Anna, "Automatic network protocol analysis." in *NDSS*, 2008.

[6] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *ACM Sigplan Notices*, 2004.

[7] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

[8] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "Dta++: dynamic taint analysis with targeted control-flow propagation." in *Proc. of Network and Distributed System Security*, 2011.

[9] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *ACM Proceedings of the 2007 international symposium on Software testing and analysis*, 2007.

[10] 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 *IEEE 37th International Symposium on Microarchitecture - MICRO*, 2004.

[11] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *ACM Sigplan Notices*, 2009.

[12] M. N. Arefi, G. Alexander, H. Rokham, A. Chen, M. Faloutsos, X. Wei, D. S. Oliveira, and J. R. Crandall, "Faros: Illuminating in-memory injection attacks via provenance-based whole-system dynamic information flow tracking," in *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).* IEEE, 2018, pp. 231–242.

[13] T. M. Cover and J. A. Thomas, *Elements of information theory.* John Wiley & Sons, 2012.

[14] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture.* IEEE Computer Society, 2004.

[15] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. Lee, "Rain: Refinable attack investigation with on-demand interprocess information flow tracking," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[16] S. Boyd and L. Vandenberghe, *Convex Optimization.* Cambridge University Press, 2004.

[17] N. Sapountzis, T. Spyropoulos, N. Nikaein, and U. Salim, "Joint optimization of user association and dynamic tdd for ultra-dene networks," *IEEE International Conference on Computer Communications (INFOCOM)*, 2018.

[18] J. Chen and A. H. Sayed, "Distributed pareto optimization via diffusion strategies," *IEEE Journal of Selected Topics in Signal Processing*, 2013.

[19] J. Mo and J. Walrand, "Fair end-to-end window-based congestion control," *IEEE/ACM Transactions on Networking*, 2000.

[20] N. Sapountzis, T. Spyropoulos, N. Nikaein, and U. Salim, "User association in hetnets: Impact of traffic differentiation and backhaul limitations," *IEEE/ACM Transactions on Networking*, 2017.

[21] A. Slowinska and H. Bos, "Pointless tainting?: evaluating the practicality of pointer tainting," in *Proceedings of the 4th ACM European conference on Computer systems*, 2009.

[22] P. Matzakos, T. Spyropoulos, and C. Bonnet, "Joint scheduling and buffer management policies for dtn applications of different traffic classes," *IEEE Transactions on Mobile Computing*, 2018.

[23] https://github.com/mnavaki/FAROS.

[24] https://www.passmark.com/.

[25] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation," in *USENIX Security Symposium*, 2004.

[26] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," 2005.

[27] G. Portokalidis, A. Slowinska, and H. Bos, "Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation," in *ACM SIGOPS Operating Systems Review*, 2006.

[28] H. Yin, Z. Liang, and D. Song, "Hookfinder: Identifying and understanding malware hooking behaviors," *CyLab*, 2007.

[29] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution." in *Proc. of Network and Distributed System Security (NDSS)*, 2008.

[30] B. Mazloom, S. Mysore, M. Tiwari, B. Agrawal, and T. Sherwood, "Dataflow tomography: Information flow tracking for understanding and visualizing full systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2012.

[31] D. A. S. de Oliveira and S. F. Wu, "Protecting kernel code and data with a virtualization-aware collaborative operating system," in *in IEEE Computer Security Applications Conference*, 2009.

[32] D. A. de Oliveira, J. R. Crandall, G. Wassermann, S. Ye, S. F. Wu, Z. Su, and F. T. Chong, "Bezoar: Automated virtual machine-based full-system recovery from control-flow hijacking attacks," in *in IEEE Network Operations and Management Symposium, 2008*, 2008.

[33] M. Dalton, H. Kannan, and C. Kozyrakis, "Deconstructing hardware architectures for security," in *5th Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD) at ISCA*, 2006.

[34] K. Piromsopa and R. J. Enbody, "Defeating buffer-overflow prevention hardware," in *5th Annual Workshop on Duplicating, Deconstructing, and Debunking*, vol. 101, 2006.

[35] J. R. Crandall and F. T. Chong, "A security assessment of the minos architecture," *ACM SIGARCH Computer Architecture News*, 2005.

[36] L. Cavallaro, P. Saxena, and R. Sekar, "On the limits of information flow techniques for malware analysis and containment," in *International conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008.

[37] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: a flexible information flow architecture for software security," in *ACM SIGARCH Computer Architecture News*, 2007.

[38] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu, "Lift: A low-overhead practical information flow tracking system for detecting security attacks," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE Computer Society, 2006.

[39] W. Hu, A. Becker, A. Ardeshiricham, Y. Tai, P. Ienne, D. Mu, and R. Kastner, "Imprecise security: quality and complexity tradeoffs for hardware information flow tracking," in *Computer-Aided Design (ICCAD), 2016 IEEE/ACM International Conference on*, 2016.

[40] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," *ACM SIGPLAN Notices*, 2015.

[41] A. Ferraiuolo, W. Hua, A. C. Myers, and G. E. Suh, "Secure information flow verification with mutable dependent types," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017.

[42] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight Jr, B. C. Pierce, and A. DeHon, "Architectural support for software-defined metadata processing," *ACM SIGPLAN Notices*, 2015.

[43] M. Ozsoy, D. Ponomarev, N. Abu-Ghazaleh, and T. Suri, "Sift: Low-complexity energy-efficient information flow tracking on smt processors," *IEEE Transactions on Computers*, 2014.

[44] A. A. De Amorim, M. Dénes, N. Giannarakis, C. Hritcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach, "Micro-policies: Formally verified, tag-based security monitors," in *IEEE Symposium on Security and Privacy (SP)*, 2015.

[45] A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach, "A verified information-flow architecture," in *ACM SIGPLAN Notices*, 2014.

[46] D. Y. Deng and G. E. Suh, "High-performance parallel accelerator for flexible and efficient run-time monitoring," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2012.

[47] M. Tiwari, X. Li, H. M. Wassel, F. T. Chong, and T. Sherwood, "Execution leases: A hardware-supported mechanism for enforcing strong non-interference," in *IEEE/ACM International Symposium on Microarchitecture*, 2009.