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

Nikolaos Sapountzis[1], Ruimin Sun[1], Xuetao Wei[2], Yier Jin[1], Jedidiah Crandall[3], and Daniela Oliveira[1]
[1] University of Florida, nsapountzis@ufl.edu, gracesrm@ufl.edu, yier.jin@ece.ufl.edu, daniela@ece.ufl.edu
[2] University of Cincinnati, weix2@ucmail.uc.edu
[3] University of New Mexico, crandall@cs.unm.edu

***Abstract*—Dynamic Information Flow Tracking (DIFT), also called Dynamic Taint Analysis (DTA), is a technique for tracking the information as it flows through a program's execution. Specifically, some inputs or data get tainted and then these taint marks (tags) propagate usually at the instruction-level. While DIFT has been a fundamental concept in computer and network security for the past decade, it still faces open challenges that impede its widespread application in practice; one of them being the indirect flow propagation dilemma: *should the tags involved in an indirect flow, e.g., in a control or address dependency, be propagated?* Propagating all these tags, as is done for direct flows, leads to overtainting (all taintable objects become tainted), while not propagating them leads to undertainting (information flow becomes incomplete). In this paper, we analytically model that decisioning problem for indirect flows, by considering various tradeoffs including undertainting versus overtainting, importance of heterogeneous code semantics and context. Towards tackling this problem, we design MITOS, a distributed-optimization algorithm, that: decides about the propagation of indirect flows by properly weighting all these tradeoffs, is of low-complexity, is scalable, is able to flexibly adapt to different application scenarios and security needs of large distributed systems. Additionally, MITOS is applicable to most DIFT systems that consider an arbitrary number of tag types, and introduces the key properties of fairness and tag-balancing to the DIFT field. To demonstrate MITOS's applicability in practice, we implement and evaluate MITOS on top of an open-source DIFT, and we shed light on the open problem. We also perform a case-study scenario with a real in-memory only attack and show that MITOS improves *simultaneously* (i) system's spatiotemporal overhead (up to 40%), and (ii) system's fingerprint on suspected bytes (up to 167%) compared to traditional DIFT, even though these metrics usually conflict.**

I. 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 during program or system execution. DIFT systems operate dynamically without requiring the availability of the source code, which makes them appealing for various types of applications, including enforcement of security policies, forensics analysis, reverse engineering and monitoring the flow of large distributed systems. Prior work has attempted to leverage DIFT mainly for privacy and security purposes. For example, some early DIFT works [1], [2], [3], [4], [5] attempted to detect different types of malware by following the information flow. Recently, DIFT has been leveraged to address different privacy and security vulnerabilities not only for modern computer operating systems (OSes), commodity software and honeypot technologies [6], [7], [8] but also for various IoT platforms [9], [10] and mobile devices [11], [12].

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 with little useful information being acquired.

While previous works have proposed some *heuristics* to tackle the problem, they usually make unrealistic assumptions to modern systems and have several limitations. For example, Panorama [1] relies on a human to manually label which indirect flows should be propagated. DTA++ [13] or DYTAN [14] rely on offline analysis requiring multiple traces, which does not scale well. RIFLE [15] and GLIFT [16] are based on static analysis, and other works have prohibitive performance overheads [7], [6]. While useful, these techniques can only partially combat the problem.

Another, not well-studied, tradeoff in modern DIFT, is the one between *semantics* and *applicability*. Most of the DIFT systems ignore semantics, to be applicable to machine code or to be scaled to whole live systems, including all processes

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.

and the kernel. For example, it is difficult to properly keep track of the flow of different semantics even after they get inserted into the system, as they usually have heterogeneous properties, different propagation speeds, and impact differently the execution context. Further, ignoring them or adapting an one-size-fits-all handling may improve the DIFT applicability, but it usually misses important knowledge about the information flow, putting a heavy toll on the DIFT performance and detection efficiency for attacks [17].

In this paper, we propose MITOS, a framework that analytically tackles the open problem of: *when an indirect flow should be propagated* in an efficient (e.g., scalable) manner. In other words, MITOS *theoretically* addresses and tackles the open problem of indirect flow propagation encountered in *practical* DIFT systems, by unifying the two, usually conflicting, worlds of theory and practice. To the best of our knowledge, this is the first work in that direction, namely to analytically study this practical problem that remains open since the past decade. Specifically, our contributions are:

(1) We *model* the open problem of optimal decisioning for indirect flow dependencies, by considering various tradeoffs encountered in practical DIFT systems such as the undertaining vs. overtainting, importance of heterogeneous code semantics and context, and we show that the complete problem is NP-hard.

(2) We relax the problem and by leveraging distributed optimization we propose an algorithm that converges to an approximately optimal solution. Specifically, it *decides* about the propagation of the indirect flows by weighting the above tradeoffs, is *of low-complexity*, *scalable*, *flexibly* adapts to different scenarios and security needs of large distributed systems.

(3) To the best of our knowledge, we are the first to introduce the *fairness* and *tag balancing* properties to the DIFT field, which control the balancing among the propagations of different tags. It matches information-theoretic intuitions about how tags should be propagated: e.g., flipping a coin that has a $50\%-50\%$ chance of heads-tails carries more information than a coin that is biased in one direction [18]. Similarly, when tag propagation becomes unbalanced towards one tag (e.g., due to the considered semantics), every object is tagged and we show that little information is gained.

(4) To assess MITOS potential in real DIFT systems, we *implemented* and *evaluated* MITOS on top of FAROS, an existing open-source DIFT system [7]. We investigated the complex tradeoffs involved in the indirect flow dilemma and we performed a case-study scenario with a real in-memory attack and showed that MITOS improved *simultaneously* (i) system's time and memory overhead (up to 40%), and (ii) system's fingerprint on suspected bytes (up to 167%) compared to standard DIFT, even though these metrics usually conflict.

The rest of the paper is organized as it follows. Section II provides basic background on DIFT. Section III discusses MITOS assumptions. Section IV details the analytical model for the indirect flow propagation problem and the corresponding solution. Section V discusses MITOS implementation and evaluation on an existing DIFT system. Section VI summarizes MITOS key findings, limitations, and future work. Section VII presents related work. Section VIII concludes the paper.

II. 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 to enable a wide variety of applications, such as enforcement of security policies, real-time forensics analysis, and reverse engineering. The main idea is based to tag 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 straight-forward, as the bytes being involved in certain system activities get tagged with some metadata. For example, in MINOS [3], an early DIFT system, all data coming from network were tagged with an extra bit indicating if the byte was suspicious. There are two types of tag propagation flows: direct and indirect.

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

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 [6], [4]. 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 with little being 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, which can

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

Fig. 1. Address dependency example.

be crucial for security applications to detect attack or violation of security policies. **While many 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 twofold: (i) to model the desired information-theoretic properties and needs of that dilemma in DIFT as an optimization problem, and then (ii) to analytically solve it using distributed optimization techniques.**

III. MITOS Assumptions

Tag differentiation. First, MITOS assumes that the DIFT 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), *process* tags (representing bytes coming from the address space of a process), or *system* tags (for different systems whose flows want to be monitored). Different tag types might come from different (sub)systems. For the sake of presentation, we denote the different tag types as: $t1, t2, ...$. Tag differentiation is a promising feature of modern DIFT systems since it captures the information flow from different perspectives [7], [8] and can efficiently visualize the information flow of large distributed systems.

Note that, depending on the DIFT system and the security or privacy application needs, MITOS is open to the consideration of any type of code semantics as soon as they get captured with different tags or tag combinations (e.g., different data types or pointer tags [17]).

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 life cycle 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 depends on the DIFT system, e.g., hashmap or duplicated memory.

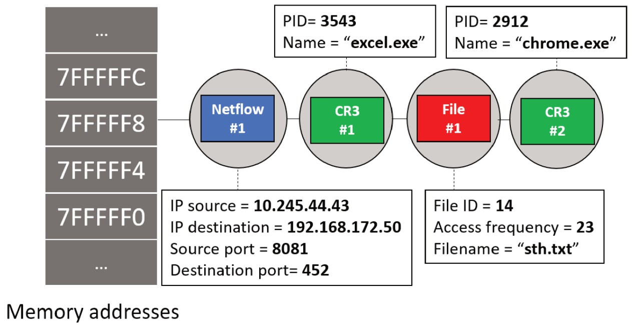


Fig. 2. Provenance list for the byte in address #7FFFFF8.

TABLE I
Notation (Inputs marked with *)

Notation	Description
t, i	tag ID
$n_{t,i}$	number of copies for the tag $\{t, i\}$
$\mathbf{n}$	2-D optimization vector (control variable vector)
α *	fairness degree in undertainting cost
β *	steepness of the overtainting cost
τ *	weight for the under/over- tainting tradeoff
u_t *	weight of tag type t while considering tag types
o_t *	weight of tag type t for the memory pollution

IV. MITOS: Decisioning Problem and Solution for Indirect Flow Propagations

This section describes MITOS theoretical framework and derived policies. MITOS' goal is to address the indirect flow propagation dilemma. Specifically, we first formulate an optimization problem for that dilemma (Section IV-A), and we then analytically tackle it through Alg.2 (Section IV-B).

A. 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 weight the different tradeoffs involved at the indirect flow propagation. Finally, (C) we define our *optimization problem*. Table I summarizes the notation used throughout the section; the indicator (*) refers to the inputs in our model.

(A. Control Variables: $\mathbf{n}$). 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 $i \in \mathbb{N}^+ = \{1, 2, 3, ...\}$ represents an integer that differentiates tags of same type. We now define as $n_{t,i} \in \mathbb{N} = \{0, 1, 2, ...\}$ 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:

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

The number of dimensions of $\mathbf{n}$ changes dynamically as the system runs, since new tags are born/deleted due to the continuous creation/ termination of processes, network connections, etc.

(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 input and 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, 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 [19], [20], [21].

(B1. Cost function of undertainting: $c_{\alpha}^{under}(\mathbf{n})$.) Now, we model the undertainting cost function. We introduce the fairness parameter $\alpha \in \mathbb{R}^{+}$ that is input and balances the number of propagations for different tags, and the parameter $u_t \in \mathbb{R}^{+}$ that weights the importance of different tag types.

$$c_{\alpha}^{under}(\mathbf{n}) = \sum_{t} u_t \sum_{i} \frac{(n_{t,i})^{1-\alpha}}{\alpha - 1}. \tag{3}$$

In the following, we discuss the properties of our considered cost function. Figure 3(a) depicts this function for different values of α. When $\alpha = 1$, the above function is not defined (as $\alpha - 1 \to 0$) and $\log (n_{t,i})^{-1}$ is used instead. Note that, the proposed α-fair fairness function was inspired by the fairness in resource allocation for wireless networks [22], [23], [19].

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 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 [17] provide more examples with different semantics that might lead to overtainting. In such scenarios, MITOS will adjust the tag propagations to prevent deterioration of 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: $\mathbf{u} = [u_{t1}; u_{t2}; ...]$. One could even consider a *tag confluence* (when two or more tags come together) to control the tag propagation of the involved tags based on a certain run context.

(B2. Cost function of overtainting: $c_{over}(\mathbf{n})$). 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. We introduce the input parameter β that dictates the slope, namely steepness, on the overtainting cost. Then,

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

In the following, we discuss the properties of our considered cost function. Figure 3(b) depicts the function of Eq. (4).

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 [24]. 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.

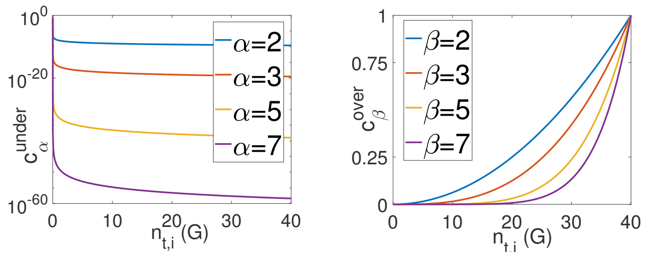
(C. Optimization Problem). Based on the defined control variables and cost function we formulate our problem.

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

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

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

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



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

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

We now explain the physical meaning of our optimization problem. 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 such a scenario is worth propagating (i.e., it improves the information flow), e.g. due to the semantics or context priorities, we propagate it and increase the corresponding value of $\mathbf{n}$, otherwise we do not (see A. Control Variables). This will become more clear in the next subsection where we describe how $\mathbf{n}$ should be best derived: $\mathbf{n}$ is updated, by leveraging the marginal costs dictated by the considered cost function (see B. Cost Function) and the considered linear constraints, every time our system encounters an indirect flow leveraging distributed optimization techniques. The marginal cost will take into consideration all the tradeoffs discussed above. Constraint Eq.(6) states that the total number of tag copies should not exceed the total tag space in all lists. Constraint Eq.(7) ensures that each tag should not have more copies than the total number of memory bytes (i.e., no byte is allowed to have more than one copy of a tag).

This problem has two main challenges at hand. First, the control variables $n_{t,i}$ of the considered vector $\mathbf{n}$ takes integer values, i.e. $n_{t,i} = 1, 2, 3, ...$, since a tag can be propagated to one or several bytes. Thus, this is a NP-hard *integer optimization problem*, which is hard to solve optimally. Second, the number of control variables $n_{t,i}$ can experience sharp increases and decreases in very short intervals (e.g., a video game reads data from files and downloads content from Internet, thus generates hundreds of file and network tags in a few milliseconds), thus continuously changing the dimensions of $\mathbf{n}$. This further complicates the problem since *the system dynamics change continuously as the system runs.*

B. 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 number of dimensions of the control variable change continuously.

Similarly to various works, we first propose to consider the continuous relaxation of the problem to obtain a closed-form real-valued solution [25], [26]. Specifically, we relax the allowed values for $n_{t,i} \in \mathbb{N}^+$ to $n_{t,i} \in \mathbb{R}^+$. This relaxation will allow us to tackle the relaxed problem fast with distributed optimization techniques.

Lemma 1. *The relaxation of $n_{t,i} \in \mathbb{N}^+$ to $n_{t,i} \in \mathbb{R}^+$ in Problem 1 transforms it in a convex optimization problem.*

Proof. The cost function is a sum of two convex functions (both second derivatives are positive), and further it is convex at the $\mathbb{R}^+$. The constraints are linear, and further convex. To this end, the relaxed problem is convex [24].

□

This relaxed problem can be solved analytically using the method of Lagrange multipliers and Karush Kuhn Tucker (KKT) conditions [24], to derive the optimal vector $\mathbf{n}^*$. [2] Note that, this solution might not scale well: (i) since new tags are created, deleted and propagated very frequently as the operating system runs posing a prohibitive overhead when centralization (e.g., tag propagation updates) and system dynamics modifications (e.g., due to the change of the control variable dimension) might need to happen too often, (ii) since the global picture (system dynamics) of all subsystems is not easily visible (accessible), especially in large distributed systems. In the following, we propose a *distributed-optimization solution* that scales well to all these scenarios.

Solution roadmap: We start with Indirect Flow Propagation (IFP) Scenario 1, where we assume that the destination's list has enough space in its provenance list to accommodate the coming tags. Then, we generalize it to IFP Scenario 2 where we investigate the problem when the space is limited, making the problem more complex.

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 $7FFFFF0 + t_3 = 7FFFFF0 + 8 = 7FFFFF8$, 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 $7FFFFF8$ that will be propagated following the basic DIFT rules (see Section II) 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?* Alg. 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 [24]. In the following two paragraphs we elaborate on the two-steps of Alg. 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

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

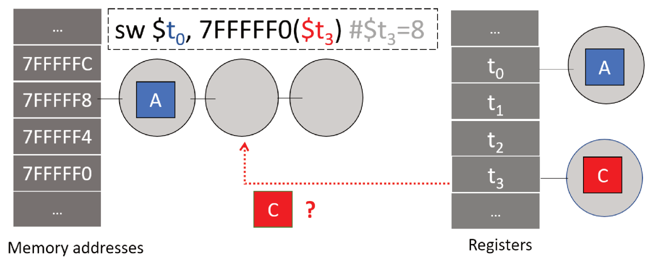


Fig. 4. Address dependency (enough space in the destination's list): the store word sw instruction copies data from the register t_0 to memory location $7FFFFF0 + t_3 = 7FFFFF0 + 8 = 7FFFFF8$.

Algorithm 1 IFP Scenario 1: Propagation of the tag with ID $\{T, I\}$ with sufficient space at the destination' provenance list.

```
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: Use the gradient-descent crit. for IFP decisioning.
4:     If $\Delta(n_{T,I}) \leq 0$ then propagate the tag $\{T, I\}$.
5:     Else, block the tag $\{T, I\}$.
```

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 $n_{T,I}$ (we use capital letters T, I to refer to a particular tag). 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:

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

The variable $\Delta n_{T,I}$ refers to the cost added by propagating that tag with ID $\{T, I\}$ 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 ($-\alpha \cdot u_t \leq 0$) of undertainting that attempts to decrease it , and on (ii) the submarginal cost ($\beta \cdot \left(\frac{\sum_t o_t \sum_i n_{T,I}}{N_R} \right)^{\beta-1} \geq 0$) of overtainting that attempts to increase it. Note that, while the former quantity differentiates for different tags (it can be derived with *local* information, in practice), the latter quantity is the same for all tags (it is actually the memory pollution, kept in a *globally* available variable for all potential subsystems, in practice). The sign of their sum, and, thus, the direction of the gradient, follows the sign of the highest absolute value.
Second, considering 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 [24]. 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 2. ***[MITOS Decisioning Rule for the IFP problem]*** *Considering the direction of the gradient of the cost function, the currently optimal rule for determining the propagation of a tag 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 (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 spaces 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 ?* We extend Algorithm 1 to Algorithm 2, to still use an adapted prioritized first-order optimization criterion.

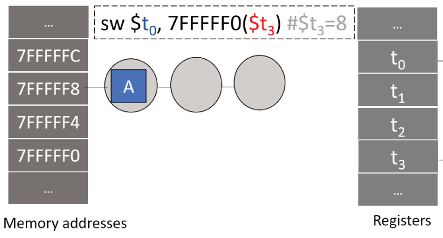


Fig. 5. Address dependency (limited space in the destination's list): the store word sw attempts to copy data from the register t_0 to memory location $7FFFFF8$.

Algorithm 2 IFP Scenario 2: Propagation of multiple tags with limited space available in the destinations' provenance list, namely A available tags space.

```
1: Derive the partial derivatives (marginal costs) for all
   involved tags in the current IFP using Eq. (8).
2: Sort the tags w.r.t. 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 2. (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).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, during each indirect flow the improvement in $c(\mathbf{n})$ is 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 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 [25], [24]. However, note that, the rule proposed might make decisions that are optimal at a given point in time, but that might push the system into a local optimum from which it cannot get out easily later. Further, the solutions of Alg. 1 and Alg. 2 do not necessarily correspond to the global optimum points of Problem 1, as they are heuristic solutions of the relaxed problems.

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

1) Our proposed rule derives the gradient of our cost function by properly *weighting* all the involved tradeoffs (e.g., undertainting versus overtainting, semantics priorities) and system dynamics (e.g., memory pollution) and then it best decides about the propagation of an indirect flow.
2) Our rule for the IFP is of *low-complexity*. The time complexity is $O(1)$, since every time MITOS needs to make an IFP decision it only needs to sum two real numbers (see Eq. (8)). For 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)).
3) It is *scalable*. MITOS only needs to retrieve a local value about how undertainted the tag is, for the IFP decisioning (and use it along with a globally available estimation of memory pollution, see e.g. Eq. 8). This keeps MITOS *scalable* as its complexity doesn't change on the number of tags in the system.
4) It is *flexible*, since by changing the input parameters one can flexibly weight the involved tradeoffs differently and capture different performance degrees based on the application scenarios and security needs. It is also α-*fair*, since α captures different degrees of tag balancing. In Section V we elaborate more on this.

MITOS can be *generalized* to capture different types of flows, processes and objectives (we discuss such a scenario in Section V-C). Due to space limitations and as the problem formulation stay the same in all cases we skip the details.

Finally, bearing these in mind, MITOS is highly efficient on large distributed systems that might need to often exchange their local information about tag propagations e.g., to detect or keep track of potential attacks that infect different subsystems, by using our low-complexity, scalable and flexible rules.

V. Applying MITOS to an Existing DIFT

To evaluate MITOS applicability in practice we implemented it in an existing, open-source DIFT system, FAROS [27], [7]. We start by detailing our implementation (Section V-A), and then 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 V-B). Finally, we evaluate MITOS in a study case, where FAROS is detecting stealthy in-memory-only attacks (Section V-C). 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 $\alpha = 1.5$, $\beta = 2$, $u_t = o_t = 1 \; \forall t \in \mathbb{T}$, $\tau = 1$, and that all τ values are normalized up to the power of 10^6. We varied the parameters in our evaluations and reached similar conclusions.

A. Implementation Details

Figure 6 illustrates our architecture which consists of five layers: (i) the host Ubuntu 14.04 machine, (ii) the QEMU virtual machine (VM) with the PANDA plugin, (iii) the open-source DIFT tool FAROS, (iv) MITOS implemented as a FAROS extension for the IFP problem, (v) Windows 7 as guest OS [3]. PANDA is built upon the QEMU whole-system emulator adding to it capabilities for instruction level analysis including *recording* and *replaying* a system run. FAROS was implemented as an PANDA plugin extension leveraging direct flow propagations for malware analysis and we refer the interested reader to [7] for details.

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

[3]FAROS was implemented for Windows 7. As discussed in Section VI, the type of OS does not affect the *nature* of the indirect flow propagation problem considered in this paper, thus the OS choice is not expected to (directly) impact the insights offered by MITOS.

at the host. Then, FAROS invokes MITOS, to propagate any indirect flow propagations (IFPs). [4] We designed the module is_IFP to filter and process the instructions that carry IFP, i.e., address or control dependencies (step (4)). Next, the instructions associated with IFP are subject to Alg. 2 (step (5)). Specifically, MITOS inspects the shadow memory and calculates the marginal costs $\Delta n_{T,I}$ for all tags T, I appeared in the source operand of the instruction, it sorts them and decides if they worth being propagated (see Alg. 2). Finally, MITOS updates the shadow memory of the propagated tags.

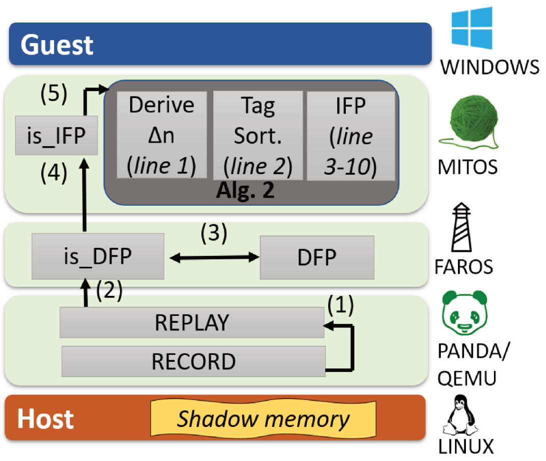


Fig. 6. MITOS implementation.

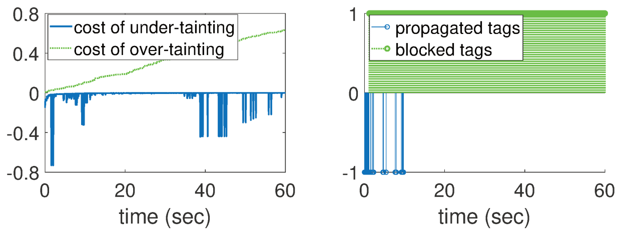
B. Sensitivity Analysis

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., for MITOS we are interested in the indirect flow decision impact, memory pollution, overhead etc). We focus on three scenarios and investigate MITOS performance on the tradeoffs involved in the indirect flow decisioning.

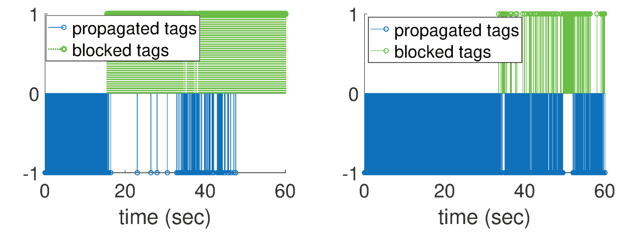
We ran an one-minute network-benchmark on Windows using the PerformanceTest tool of Passmark [28], where the guest acts as a client and downloaded several megabytes of data from a remote server. We replayed this record multiple times with MITOS on top of FAROS using different values for our inputs (see Table I). Below, we focus on the impact of our inputs on the tradeoffs involved on IFP decisioning.

Undertainting vs. overtainting. The parameter that weights this tradeoff is $\tau \in \mathbb{R}^+$, where the higher the τ the more emphasis is put on the cost of overtainting. Fig. 7 shows how the system reacts for three different values of τ. We replayed the one-minute recordings three times, using different values of $\tau = 1, 10^{-1}, 10^{-2}$, keeping all other parameters fixed, and waited 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 (y-axis) for different indirect flow propagations that MITOS encountered as a function of time (x-axis) 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 over 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 we 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 τ decrease.

Message: The undertainting vs. overtainting tradeoff should not be a challenge in the indirect flow dilemma and DIFT efficiency. MITOS appropriately weights it with respect to τ and best decides leveraging a simple propagation rule (see Lemma 2).



(a) time vs. marginal costs, $\tau = 1$. (b) time vs. IFP decisions, $\tau = 1$.



(c) time vs. IFP decisions, $\tau = 0.1$. (d) time vs. IFP decisions, $\tau = 0.01$.

Fig. 7. (a): [x-axis] time vs. [y-axis] marginal costs (see Eq. 8). (b)-(d): [x-axis] time vs. [y-axis] IFP decisions (we plot +1 for the blocked and -1 for the propagated IFP., see Alg. 2)

Fairness - Tag balancing. The higher the α, the more fair MITOS is (see Eq. (3)). Specifically, as $\alpha \rightarrow \infty$ the undertainting cost becomes steeper, i.e. it penalizes more intensely the overpropagated tags. This attempts to maximize the tags with fewer copies to improve tag balancing. Fig. 8 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

[4] MITOS can also track DFP: we investigate it in a study case in Section V-C. There, (i) the functions is_DFP and DFP are removed from FAROS, and (ii) both direct and indirect flows are forwarded to MITOS and further to Alg. 2. To do so, we replace the function is_IFP with $is\_DFP\_or\_IFP$, i.e. MITOS now handles instructions related to both direct and indirect flows.

the number of copies of different tags. The sharp deviations of the tags can be alleviated by adapting α, thus improving tag balancing performance, and entropy, up to $2\times$. This is important as traditional DIFT systems tend to overpropagate tags in multiple scenarios, consequently hurting their overall performance and wasting memory resources from the provenance lists [7].

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 these systems experience situations where they tend to overpropagate certain tags, especially in large distributed systems that the global picture is not easily visible.

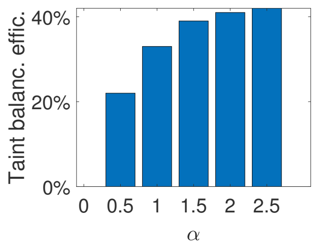


Fig. 8. α vs. fairness and tag-balancing.

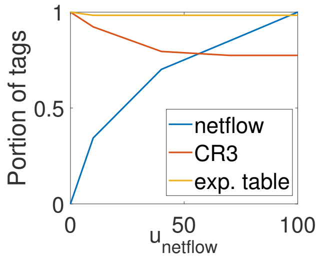


Fig. 9. u_{netflow} vs. propagated netflow tags.

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 [7]. As discussed, there are many reasons that warrant dynamic and somewhat personalized strategies to determine the weight of over vs under tainting based on tag type, through u_t. In Fig. 9 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 percentage of netflow tags, encountered at the end of each replay, 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 (export table tags have higher undertainting cost and further are mildly decelerated).

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.

Finally, we also ran CPU and file-system benchmarks, and we noticed similar behaviors. We skip the results for those benchmarks due to space limitations.

C. *Case study: Flagging In-Memory-Only Attacks*

We now apply MITOS in FAROS while it is flagging stealthy *in-memory 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 the attack compared to the standard FAROS (time complexity), how much memory is used (space complexity), and how many bytes can successfully be detected as suspicious (detection efficiency) in each case.

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

TABLE II
TIME, SPACE COMPLEXITY, AND NUMBER OF BYTES THAT WERE SUCCESSFULLY DETECTED IN AN IN-MEMORY ATTACK.

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 comes from the Internet and is associated with *netflow* tag. 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 [7]. We set up the attacker's VM (Linux Kali) and generated a shell code that ran in the victim's VM (Windows 7). This opened a session for the attacker and we then perform a remote reflective DLL injection targeting the victim process *calculator.exe*. As explained earlier, we consider two systems and attempt to compare their performance: (i) [FAROS] propagating aggressively all direct flows and no indirect flows as suggested in various DIFT systems including FAROS, and (ii) [MITOS] propagating all flows (direct and indirect) at the MITOS level. For (ii) we generalize MITOS to also consider direct flows, as explained in Sections V-A.

The spatiotemporal complexity and the detection performance of both systems is depicted in Table II. We ran six Metasploit shells (reverse https, reverse https proxy, reverse tcp rc4 dns, reverse tcp rc4, reverse tcp) and show the average performance. While FAROS aggressively propagates all tags [7], MITOS propagates only the tags that are important based on our considered objective that measures the information-flow (see Alg. 2). *We note that MITOS achieves the following improvements **simultaneously** (even though they usually come in an antagonistic fashion): (i) it propagates fewer tags than FAROS by further alleviating the space and time needed for the tracking analysis* $1.65\times$ *and* $1.11\times$ *respectively, and (ii) it can successfully detect* $2.67\times$ *times more bytes that were involved in the in-memory attack.*

Message: MITOS opens new horizons of how information flow can be measured and optimized in modern and large systems where spatiotemporal complexity emerges as a key performance bottleneck.

VI. DISCUSSION AND FUTURE WORK

MITOS consists of an analytical algorithm and set of policies for the open problem of indirect flow propagation in modern DIFT systems. *From a theoretical viewpoint*, MITOS is novel as it analytically models the tradeoffs between undertainting and overtainting and between the importance of heterogeneous code semantics and context, and designs a solution algorithm using distributed optimization. It also introduces fairness and tag-balancing: two key properties for the success of modern DIFT, and investigates the impact of different α-fair degrees on system performance. *From a systems viewpoint*, MITOS is distributed, scalable, flexible, of low complexity, applicable to large distributed systems. In our extensive performance evaluation we demonstrated that it can be easily applied to an existing software-based DIFT system, and then shed some light on the various dimensions of the open problem. Additionally, we performed a study case scenario with an in-memory-only attack. There, we showed that MITOS can improve simultaneously spatiotemporal complexity up to 40% and the detection efficiency up to 167%, compared to traditional DIFT. Summing up, *MITOS analytically studies and addresses a problem of DIFT systems that has been open for the last decade: the problem of whether indirect flows should be propagated or not; a dilemma that impedes the application of DIFT to practice. Then, we demonstrate our decisioning framework under an existing DIFT system.*

Scheduling management in the lists. In our evalation, we followed FAROS assumptions [7] and 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 [25].

MITOS 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 OS 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. 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 OS 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.

Limitations. 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. Finally, note that FAROS runs on Windows 7, restricting our OS choice for our case study analysis. While the OS itself does not affect the nature of the open IFP problem and the insights offered by MITOS, we plan to also apply MITOS in more modern OSes in our future work.

VII. Related Work

DIFT in the context of detecting attacks was co-introduced by Costa et al. and Suh et al. [4], [29]. TaintBochs [30] was another early application of taint analysis to analyze data lifetime in a full system. Other early DIFT works include [3], [2], [31] that explore policy tradeoffs for DIFT schemes and higher-level systems issues, vulnerabilities in commodity software and honeypot technologies. Malware analysis [1], [32], network protocol analysis [33] and data flow tomography [7], [34], full-system recovery after memory corruption attacks and protecting kernel integrity against rootkits [35] 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 that identified the problems with address and control dependencies include [4], [3]. For Minos, it is claimed that (i) address dependencies are propagated for 8- and 16-bit loads and stores, and blocked for 32-bit loads and stores. More details and analysis of these issues followed [36], [5], including a quantitative analysis of full-system pointer tainting [17]. More recent DIFT systems that are designed for flexibility [15], [1] enable address and/or control dependencies to be tracked based on a user-provided policy, but provide no satisfactory policy for doing so in practice. As discussed earlier, Panorama [1], DTA++ [13], DYTAN [14],RIFLE [15], and GLIFT [16] suffer from several limitations when it comes to indirect flow propagation and overtainting. Panorama [1] relies on a human to manually label which address and control dependencies tags should be propagated. Address and control dependencies arise from common program structures, such as conditional statements, for loops, and arrays. DTA++ [13] or DYTAN [14] rely on off-line analysis, which does not scale to full systems. Systems designed with correctness as the primary goal, such as RIFLE [15], and GLIFT [16], propagate all tags all the time unless a compiler statically analyzes the information flow. While there are attempts at implementing DIFT in hardware [37], [38] they do not handle indirect flows and taint analysis performance overheads.Other recent DIFT schemes include FAROS [7] and V-DIFT [6] that approximate a quantitative information flow at the price of overhead, not scaling to large systems. DDIFT [39] considers generic direct and indirect tag propagation policies for IoT.

VIII. Conclusion

In this paper we analytically study the problem of indirect flow propagation encountered in practical DIFT systems, by unifying two, usually conflicting, worlds of theory and practice. After modeling the open problem, we propose MITOS, an iterative, scalable and distributed algorithm to tackle it, implementable in large systems. We then evaluate it in an existing software-based DIFT by shedding light on the problem of indirect flow propagation, investigate the complex tradeoffs involved, and we show the significant performance improvement (e.g., 167% in our case scenario).

IX. Acknowledgements

We would like to thank the anonymous reviewers for their insightful comments. This research has been supported by DARPA Trusted Computing Project grant No. FA8650-15-C-7565 and NSF CNS 1801599, NSF CNS 1801613, and NSF CNS 1801593.

References

[1] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *ACM CCS*, 2007.

[2] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software." in *NDSS*, 2005.

[3] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *IEEE/ACM MICRO*, 2004.

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

[5] J. R. Crandall and F. T. Chong, "A security assessment of the minos architecture," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 1, pp. 48–57, 2005.

[6] 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 ARES*, 2016.

[7] 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 *IEEE/IFIP DSN*, 2018.

[8] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. Lee, "Rain: Refinable attack investigation with on-demand inter-process information flow tracking," in *ACM CCS*, 2017.

[9] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel, and A. S. Uluagac, "Sensitive information tracking in commodity iot," in *USENIX Security Symposium*, 2018.

[10] I. Bastys, M. Balliu, and A. Sabelfeld, "If this then what?: Controlling flows in iot apps," in *ACM CCS*, 2018.

[11] 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," *OSDI*, 2010.

[12] B. Gu, X. Li, G. Li, A. C. Champion, Z. Chen, F. Qin, and D. Xuan, "D2taint: Differentiated and dynamic information flow tracking on smartphones for numerous data sources," in *IEEE INFOCOM*, 2013.

[13] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "Dta++: dynamic taint analysis with targeted control-flow propagation." in *NDSS*, 2011.

[14] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *ACM ISSTA*, 2007.

[15] 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/ACM MICRO*, 2004.

[16] 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*, vol. 44, no. 3, 2009, pp. 109–120.

[17] A. Slowinska and H. Bos, "Pointless tainting?: evaluating the practicality of pointer tainting," in *ACM EuroSys*, 2009.

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

[19] N. Sapountzis, T. Spyropoulos, N. Nikaein, and U. Salim, "Joint optimization of user association and dynamic tdd for ultra-dene networks," *IEEE INFOCOM*, 2018.

[20] N. Sapountzis, T. Spyropoulos, N. Nikaein, and U. Salim, "An analytical framework for optimal downlink-uplink user association in hetnets with traffic differentiation," in *2015 IEEE GLOBECOM*, 2015.

[21] ——, "Optimal downlink and uplink user association in backhaul-limited hetnets," in *IEEE INFOCOM*, 2016.

[22] J. Mo and J. Walrand, "Fair end-to-end window-based congestion control," *IEEE/ACM Transactions on networking*, vol. 8, no. 5, pp. 556–567, 2000.

[23] 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*, vol. 25, no. 6, pp. 3396–3410, 2017.

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

[25] 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.

[26] L. Vigneri, T. Spyropoulos, and C. Barakat, "Low cost video streaming through mobile edge caching: Modelling and optimization," *IEEE Transactions on Mobile Computing*, 2018.

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

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

[29] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-end containment of internet worms," in *ACM SOSP*, 2005.

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

[31] 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.

[32] H. Yin, Z. Liang, and D. Song, "Hookfinder: Identifying and understanding malware hooking behaviors," *in NDSS*, 2008.

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

[34] 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)*, vol. 9, no. 1, p. 3, 2012.

[35] D. A. S. de Oliveira and S. F. Wu, "Protecting kernel code and data with a virtualization-aware collaborative operating system," in *in ACSAC*, 2009.

[36] M. Dalton, H. Kannan, and C. Kozyrakis, "Deconstructing hardware architectures for security," in *WDDD*, 2006.

[37] 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 *IEEE/ACM ICCAD*, 2016.

[38] A. Ferraiuolo, W. Hua, A. C. Myers, and G. E. Suh, "Secure information flow verification with mutable dependent types," in *DAC*, 2017.

[39] N. Sapountzis, R. Sun, and D. Oliveira, "Ddift: Decentralized dynamic information flow tracking for iot privacy and security," in *Workshop on Decentralized IoT Systems and Security (DISS)*, 2018.