Challenges and Opportunities for Practical and Effective Dynamic Information Flow Tracking

CHRISTOPHER BRANT, University of Florida, USA
PRAKASH SHRESTHA, University of Florida, USA
BENJAMIN MIXON-BACA, Arizona State University, USA
KEJUN CHEN, University of Florida, USA
SAID VARLIOGLU, University of Cincinnati, USA
NELLY ELSAYED, University of Cincinnati, USA
YIER JIN, University of Florida, USA
JEDIDIAH CRANDALL, Arizona State University, USA
DANIELA OLIVEIRA, University of Florida, USA

Information flow tracking was proposed more than 40 years ago to address the limitations of access control mechanisms to guarantee the confidentiality and integrity of information flowing within a system, but has not yet been widely applied in practice for security solutions. Here we survey and systematize literature on dynamic information flow tracking (DIFT) to discover challenges and opportunities to make it practical and effective for security solutions. We focus on common knowledge in the literature and lingering research gaps from two dimensions – i) the layer of abstraction where DIFT is implemented (software, software/hardware, or hardware) and ii) the security goal (confidentiality and/or integrity). We observe that two major limitations hinder the practical application of DIFT for on-the-fly security applications: i) high implementation overhead and ii) incomplete information flow tracking (low accuracy). We posit, after review of the literature, that addressing these major impedances via hardware parallelism can potentially unleash DIFT's great potential for systems security, as it can allow security policies to be implemented in a built-in and standardized fashion. Furthermore, we provide recommendations for the next generation of practical and efficient DIFT systems with an eye towards hardware-supported implementations.

CCS Concepts: • **Security and privacy → Information flow control**; • **General and reference → Surveys and overviews**.

Additional Key Words and Phrases: Dynamic Information Flow Tracking, Dynamic Taint Analysis, Confidentiality, and Integrity

ACM Reference Format:
Christopher Brant, Prakash Shrestha, Benjamin Mixon-Baca, Kejun Chen, Said Varlioglu, Nelly Elsayed, Yier Jin, Jedidiah Crandall, and Daniela Oliveira. 2021. Challenges and Opportunities for Practical and Effective Dynamic Information Flow Tracking. *ACM Comput. Surv.* 1, 1, Article 1 (August 2021), 33 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Authors' addresses: Christopher Brant, University of Florida, USA; Prakash Shrestha, University of Florida, USA; Benjamin Mixon-Baca, Arizona State University, USA; Kejun Chen, University of Florida, USA; Said Varlioglu, University of Cincinnati, USA; Nelly Elsayed, University of Cincinnati, USA; Yier Jin, University of Florida, USA; Jedidiah Crandall, Arizona State University, USA; Daniela Oliveira, University of Florida, USA.

1 INTRODUCTION

Although classic access control mechanisms (via authentication and authorization) constrain the access rights of a user, they do not guarantee the confidentiality and integrity of the information flowing within a system or a program [9]. To address this limitation, information flow models have been introduced in the 1970s [25, 26, 31, 32] with the key idea to taint certain instructions and/or data with metadata tags, to propagate these tags based on a given confidentiality and/or integrity policy, and to check these tags for specific policy(ies) compliance. Several techniques for tracking information flow have been proposed, which fall into two major categories – *static* and *dynamic*. Static approaches analyze a program prior to its execution (via source code) to verify all possible execution paths. Dynamic approaches, on the other hand, control a program or system information flow at runtime.

In the 2000s, information flow tracking was revisited as a promising approach to mitigate control-flow hijacking attacks (e.g., buffer overflows) [19–22]. The research community at that time was particularly interested in dynamic mechanisms, or Dynamic Information Flow Tracking (DIFT), also denoted as Dynamic Taint Analysis (DTA) [66]. Since then, several DIFT implementations for security have been proposed for – i) software, e.g., [19, 25, 26, 35, 52, 53, 56, 82], ii) hardware, e.g., [46, 47, 55, 58, 68, 79, 80], and iii) a combination of the two, e.g., [21–23, 57, 61, 75, 76, 80].

Besides detection of control flow exploitation and in-memory-only injection attacks [4] (both to assure integrity), DIFT has also been proposed to detect information leakage [59] and to locate cryptographic keys in memory [30] (both to assure confidentiality).

DIFT has also been leveraged for non-security applications, such as program visualization [51, 54] and performance improvement and benchmarking [40, 49, 67]. Despite being a relatively mature computer science theory and a highly researched area since the mid-2000s, DIFT has minimal adoption in real-world applications to protect the security of computer systems and the privacy of their users for two main reasons, described below.

1. High Implementation Overhead: Several implementations of DIFT, particularly those in software and encompassing the whole system, incur significant performance and memory overheads (e.g., 56x performance slowdown for a QEMU based implementation [4]). Although software-based DIFT offers high flexibility for implementation (e.g., tag sizes, format, and granularity), tailoring to a variety of attacks and applications, the incurred performance costs make these systems impractical for real world deployment [4]. This limitation called for investigating DIFT implementations in hardware [46, 47, 55, 58, 68, 79, 80], as envisioned by early discussions by Denning [25, 26] and Fenton [31, 32], or via software-hardware collaboration [23, 57, 61, 75, 76, 80]. These hardware-oriented approaches have shown promising reduction in performance overhead in comparison to their software-based counterparts, but they tend to lack flexibility and require significant modifications in the hardware (and even in the software) subsystem.

2. Low Accuracy (incomplete tracking of all information flows): Many DIFT systems only address direct information flows (more details in Section 2) because these flows are straightforward to detect and such an approach leads to lower performance and memory overhead in taint tracking operations (e.g., fewer tags and less tag metadata). This can lead to undertainting because some information flows (e.g., the predicate in a conditional statement) might not be captured. To be sound, a DIFT solution will taint both direct *and* indirect flows. Unfortunately, this can lead to potentially extraneous tag propagation, and can lead to taint explosion (overtainting) and high performance/memory costs. While some works [4, 30, 65, 69] have attempted to address this dilemma, the problem is still open.

Recognizing a high potential for real-world and practical implementations, in this paper we survey and systematize the literature on *dynamic* information flow tracking, with the goal to discover challenges and opportunities to make DIFT practical. Specifically, we look at the common knowledge in the field and attempt to discern research gaps from

two dimensions. First, *the layer of abstraction for DIFT implementation*, i.e., software, hardware, and hardware-software (via collaboration between these layers). Second, the *security goal*, i.e., confidentiality and/or integrity. We also provide recommendations for the next generation of efficient hardware implementations of DIFT, such as investing in hardware parallelism to allow addressing of both performance and incomplete tracking of all information flows dilemma.

To the best of our knowledge, the only work that attempted to systematize DIFT based on the literature of the time is that of Schwartz et al. [66], which is now a decade old, and does not consider implications of the layer of abstraction for the DIFT implementation nor the policy models, two key dimensions of DIFT considered in our paper. Moreover, in this present paper, we introduce a comprehensive coverage and systematization of hardware proposals for DIFT not yet covered by prior work. Our work makes the following scientific contributions:

(1) We systematize the DIFT literature with an eye towards making it practical and effective for real-world deployment from two major dimensions – i) the layer of abstraction for implementation, i.e., software, hardware, and hardware-software and ii) the security goal, i.e., confidentiality and/or integrity.
(2) We provide an exposition of common knowledge and formulate the research gaps in the existing literature.
(3) We provide recommendations for the next generation of efficient and effective DIFT systems that can bridge the research gaps identified in our survey.

2 BACKGROUND

Information flow refers to the way information moves within a system or program [25]. Research on the topic started in the 1970s with the definition of the problem and the realization that authentication methods alone cannot guarantee the confidentiality and integrity [25, 26, 31, 32] of a system or program. Initial research centered on the definition of policies designed to protect the confidentiality and/or integrity of information by determining the way the information moves throughout the system.

2.1 Information Flow Model

An information flow model can be viewed as an extension to the concept of state machine that comprises objects, state transitions, and flow policies (or lattice states) [34]. Entities involved in information flows are categorized into either *object* or *subject*. An object denotes the location where information is stored, e.g., files, memory segment, program variables, etc. A subject denotes an entity that causes (accesses) the flow of information from an object to another object, e.g., user, system, program, process, etc. More formally, an information flow model (FM) [34] is defined by a quintuple [25], i.e., $FM =< N, P, SC, \oplus, \rightarrow >$, where:

- $N = \{a, b, \dots\}$ is a set of logical storage *objects* (e.g., files, memory segment, program variables, or a system user).
- $P = \{p_1, p_2, \dots\}$ is a set of *processes* (or any active agents) that are responsible for all information flows.
- $SC = \{A, B, ..\}$ is a set of *security classes* corresponding to information that intends to cover the notion of "security classification", "security categories", and "security clearance". Each object 'a' is bound to a security class, denoted by $\underline{a}$, which indicates the security class of the information stored in the object 'a'. Users can be bound to security classes, referred to as "security clearances". Processes can be bound to security classes, which may be determined based on the security clearance of their owners or the history of security classes to which they have had access. There are two methods to bind objects to security classes – *static binding*, where the security class of an object remains constant throughout its life cycle and *dynamic binding*, where the security class of an object changes based on its content.

- '$\oplus$' is a binary operator that combines the security classes of operands when any binary function is applied on those operands. In other words, '$\oplus$' specifies the security class of the result obtained by applying a binary function based on the operands' classes. The operator '$\oplus$' is associative and commutative in nature. Given this, the class of the result of n-ary function $f(a_1, a_2, a_3, \ldots, a_n)$ is $\underline{a}_1 \oplus \underline{a}_2 \oplus \underline{a}_3 \oplus \ldots. \oplus \underline{a}_n$. Furthermore, the operator $\oplus$ is independent of the function that is used to combine values.
- '$\rightarrow$' indicates a flow relation between a pair of security classes. For classes A and B, $\underline{A} \rightarrow \underline{B}$ indicates that information is permitted to flow from class A to class B. If information flows from a function $f(a_1, a_2, a_3, \ldots a_n)$ to an object b that is statically bound to security class $\underline{b}$, $\underline{a}_1 \oplus \underline{a}_2 \oplus \underline{a}_3 \oplus \ldots. \oplus \underline{a}_n \rightarrow \underline{b}$ must hold. If b is a dynamically bound object, then for the above information flow, the class of b, i.e., $\underline{b}$, must be updated (if necessary) to satisfy $\underline{a}_1 \oplus \underline{a}_2 \oplus \underline{a}_3 \oplus \ldots. \oplus \underline{a}_n \rightarrow \underline{b}$.

The goal of the information flow model is to prevent unauthorized and/or insecure flows of information in any direction. An information flow model is typically implemented to enforce *confidentiality* and/or *integrity*, two of the three pillars of the CIA triad. Confidentiality policies aim to prevent the flow of information to an unauthorized entity, while integrity policies intend to restrict unauthorized modification of data. The *Bell-LaPadula* model [7, 45] is the most prominent model to address system security from a confidentiality perspective, while the *Biba* model [8] is the most distinguished to address systems integrity. We here summarize both models.

2.1.1 ***The Bell-LaPadula Model [7, 45].*** Intends to preserve confidentiality when information flows from one secure state to another – "*read-down, write-up*". Specifically, a subject at a confidentiality level can only access (or read) an object of lower confidentiality level. Also, a subject at a given confidentiality level can send (or write) information to an object of higher confidentiality level, often referred to as *Star (*) Security Property*. For an instance, a layperson, say `Mary`, can send (or write) sensitive information to the Federal Bureau of Investigation (FBI) because it has higher confidentiality level than her.

2.1.2 ***The Biba Model [8].*** Focuses on the principle of data integrity and is the dual of the Bell-LaPadula's model. The Biba model intends to prevent a subject from corrupting data in a level ranked higher than that of the subject –"*read-up, write-down*". A subject at a given level of integrity is only permitted to access (or read) an object of equal or higher integrity level. Similarly, a subject at a given level of integrity is only permitted to write to an object of higher integrity level. For an example, only designated scientists and medical doctors can formulate guidelines to prevent pandemics. These guidelines, in turn, can be read by people with a lower integrity label, e.g., a layperson like `Mary`. However, a layperson should not write pandemic guidelines for the public.

2.2 Types of Flows

Given a program with a sequence of commands, information flows from an object to another if the initial information stored in the former affects the information of the latter when the program is executed [9]. Information flows can be divided into two major categories – i) *direct flows* and ii) *indirect flows*. In direct flows, the information in some variable is explicitly moved to another. For example, in a command `y = x;`, information flows from the variable x to the variable y. In indirect flows, the program's control flow or address indexing (e.g., into a data structure) affects the information in a variable to another. For example, in the snippet of code below, information flows from x to y without explicit assignment of the form $y = f(x)$, where $f(x)$ is an arithmetic expression with the variable x. Here, the information stored in x defines the information of y.

```
if (x == 1)
    y = 0;
else
    y = 1;
```

The work by Denning [25], however, does not consider two other important types of information flows that have been exploited in recent attacks: *covert channels* and *side channels*. In contrast to the channels that carry direct and indirect flows, covert and side channels are channels not explicitly designed for valid communication and information exchange. The main distinction between side channels and covert channels is whether the sender is in collusion with the receiver. An example of exploiting a side channel would be an adversary stealing bits of information from a privileged process because of the manner the process uses the cache, and not because the process intends to be sending bits out of a proper channel. Conversely, if that process is leaking the bits to the adversary on purpose (i.e., colluding with the adversary), information is flowing through a covert channel.

2.3 Information Flow Tracking Mechanisms

Techniques for tracking information flow fall into two categories, as described below.

2.3.1 Static Mechanisms. Operate by examining whether a program's information flows violate a security policy *prior to* its execution, typically during program compilation. Static approaches require the programmer to associate security classes for all the objects referenced in the system. Since information flow analysis is performed prior to execution, static mechanisms do not impair the program's execution speed. Furthermore, the information flow examination process can be specified in terms of high-level language, rather than low-level hardware instructions. However, static mechanisms cannot examine (or verify) flows that are not specified by the program (e.g., language-specific defects, such as not checking array bounds, occurrences of dangling references). Moreover, a program marked secure by static analysis might be vulnerable to hardware defects and/or vulnerabilities [25]. Nevertheless, static mechanisms are often used to detect vulnerabilities in the program source code (e.g., vulnerabilities in the IoT backend architecture [3, 36]) prior to its execution.

2.3.2 Dynamic Mechanisms. Operate by examining whether information flowing within a program execution conforms to the established security policies at runtime. Examining an direct flow, e.g., $b = f(a_1, a_2, \ldots, a_n)$, is straightforward. In direct flows, prior to the execution of the assignment, runtime mechanisms verify $\underline{a}_1 \oplus \underline{a}_2 \oplus \ldots \oplus \underline{a}_n \rightarrow \underline{b}$, where $\oplus$ is a binary operator combining (according to a policy) the security classes of the operands and $\rightarrow$ indicates a flow relation between a pair of security classes. If the condition is satisfied according to the established security policy(ies), the assignment succeeds, otherwise, it fails. Since dynamic analysis follows program control flow, information flows that are not taken during the program execution are ignored. Fenton [31, 32] explored this challenge and introduced a special abstract machine described below (*Data Mark Machine*).

2.4 Data Mark Machine

Fenton [31, 32] introduced an interesting runtime policy enforcement mechanism, called *Data Mark Machine*. In Fenton's machine, each object is bound with a security class, often called tag or data mark. One important proposition of Fenton's machine was to also bind the Program Counter (PC) with a security class. Given that conditional statements (or branches) in a program are merely assignments to the PC, binding of PC with a security class enables the machine to treat implicit

flows as direct flows. Of note: we will discuss in Section 4.4.2 that the term indirect flows is a property of dynamic information flows, while the term implicit flows is used for static analysis of information flows. Fenton's machine, described below, is fundamentally a static approach. The same rationale is valid for the use of direct vs. explicit flows.

Consider a conditional structure $c : S_1, S_2, S_3, \ldots, S_m$ conditioned on the values of k conditioned variables $c_1, c_2, c_3, \ldots, c_k$. Immediately prior to the execution of the conditional structure c, the machine pushes PC along with its current security class $\underline{PC}$ to the program stack, replaces PC with the least upper bound of its own class $\underline{PC}$ and classes of the k condition variables, i.e., $\underline{PC} = \underline{PC} \oplus \underline{c}_1 \oplus \underline{c}_2 \oplus \underline{c}_3 \oplus \ldots \oplus \underline{c}_k$. Once the conditional statement(s) S have been executed, the machine restores the PC and $\underline{PC}$ by popping the top element and its security class from the program stack.

If statement S denotes an explicit flow from object $a_1, a_2, a_3, ..a_n$ to statically bound object b, the execution machine verifies if $\underline{a}_1 \oplus \underline{a}_2 \oplus \ldots \oplus \underline{a}_n \oplus \underline{PC} \rightarrow \underline{b}$ is satisfied, and prevents execution if the condition is not satisfied. If b is a dynamically bound object, the class of b is updated to satisfy $\underline{a1} + \underline{a_2} + \ldots + \underline{a_n} + \underline{PC} \rightarrow \underline{b}$. In other words, the execution-based mechanism forces the relation to be true by updating the class of b, i.e., $\underline{b}$. Fenton's machine examines implicit flows in a way similar to that of explicit flows, since the inclusion of the security class to the PC allows to treat implicit flows as explicit flows.

Fenton defined five instructions – *increment, conditional, return, branch*, and *halt* instructions. The snippet (taken from [9]) below shows an example of conditional instruction and its execution equivalent to classes of variables. *skip* denotes that the statement is not executed, $push(x, \underline{x})$ denotes to push variable x and its security class $\underline{x}$ onto the program stack, $pop(x, \underline{x})$ denotes to pop the top value and its security class and assign them to x and $\underline{x}$, respectively. Please refer to Bishop [9][p.562-565] for further details on the relationship between remaining instructions and their execution equivalent of classes of variables.

```
if (x = 0) {
    push (PC, PC);
    PC = (PC, x);
    PC = n;
}
else
     if PC <= x
         x = x - 1;
```

Both in Fenton's seminal journal paper [32] and in his dissertation [31], Fenton describes a side channel that completely subverts the security guarantees of the Data Mark Machine if the system has variable data marks for registers. To compile a program for Fenton's Data Mark Machine, or to write an assembly language program for it, the compiler or the program's author must effectively analyze the program for all possible flows of information statically to know, for every program point, what register (based on its mark) data can go into. The register and data must have the same mark. Thus, the Data Mark Machine itself acts only as a (perhaps redundant) runtime check to ensure that the static analysis of information flow does not violate the security of the system. Fenton's Data Mark machine was more of a thought experiment about the difficulties of having a perfect DIFT system than a practical system that could actually track information flow in a useful way.

While it is tempting to treat DIFT as a special case of other information flow techniques and controls, DIFT is a distinct technique with problems that are analogous to, but not the same as, the problems that arise from other information

controls. DIFT is distinguished from Denning's Lattice Model [25] not only in that the approach is dynamic rather than static, but also in that DIFT operates on a trace and typically assumes no access to the source code, while Denning's Lattice Model operates on programs and assumes access to the full source code. DIFC (Decentralized Information Flow Control) [44, 53] has an acronym that is similar to DIFT and sometimes involves run-time checks, but is fundamentally a static approach that works on programs and requires access to the full source code.

3 SYSTEMATIZATION METHODOLOGY

The focus of our work is systematizing the existing and prominent literature on DIFT after the active revisiting of the field in the mid-2000s. Specifically, our systematization focuses on critical aspects of DIFT systems that enhance (or constrain) its practicality. The goal of our work is to elucidate common knowledge on DIFT, its applications, and the types of implementations in different systems, as well as research gaps that hinder DIFT deployment. Another goal of our systematization is to identify opportunities for enhancement of DIFT implementations in software, hardware, or their combinations, so that it becomes practical for assuring the confidentiality and integrity of general computer systems.

3.1 Research Questions

To aid in the identification, classification, and analysis of the literature used in our systematization, our process was directed by the following research questions (RQ):

(1) What are the key attributes of a DIFT system that influence its performance and accuracy?
(2) What is the state of the art for DIFT solutions implemented exclusively in software?
(3) What is the state of the art for DIFT solutions implemented by leveraging both the software and hardware subsystems?
(4) What is the state of the art for DIFT solutions implemented exclusively in hardware?

The systematization process followed the guidelines for systematization of knowledge papers by Budgen et al. [11] and Kitchenham et al. [43]. We systematized the literature on DIFT that we incorporated in this work following a general, three-phased process - (i) *searching*, (ii) *filtering*, and (iii) *categorization*, as explained in the next subsections.

3.2 Searching and Identifying Relevant Works

The initial step for the identification of relevant related works was for us to identify specific keywords related to DIFT, and to search for those keywords in academic databases or reputable publishers in computer science and engineering. The five major places we originally searched were the IEEE, ACM, USENIX, ISCC, and Springer academic databases. Specifically, our base keywords utilized were "information flow", "DIFT", and "taint tracking".

Additionally, for each keyword, subsequent searches were performed individually appending each of the terms: "Indirect Flows", "Coprocessor", "Hardware", "IoT", and "Applications". All papers considered in our systematization were peer-reviewed.

3.3 Manually Filtering and Selecting Relevant Works

Next, six researchers (co-authors) manually reviewed the papers and removed any paper that was not directly relevant to DIFT. To pare down the set of works that were initially collected for use in our study, we manually filtered and selected relevant works according to the following criteria:

- **Inclusion Criteria.** The article or work should include theory, practice, methods, implementations, limitations, or applications regarding Dynamic Information Flow Tracking or Dynamic Taint Analysis. Additionally, relevant works should include either discussions of the fundamental practices and methods of DIFT or descriptions of implementations, methods, or techniques for DIFT. Furthermore, the relevant works should be unique in their scope of discussing DIFT or should discuss a novel method or technique for implementing DIFT. Lastly, most articles should be published between the years 2000-2021.
- **Exclusion Criteria.** Articles that do not describe any unique method, technique, implementation, nor provide any new insight into theory or analysis of DIFT. Any article or work that does not have any relevance to our research questions or background on DIFT. We also filtered articles that proposed an information flow technique that exclusively relied upon static information flow tracking or the ability to have prior knowledge of the program source code to determine vulnerabilities. An example of an excluded paper is Timber-V [81], because although the proposal involved hardware tags, there was no concept of tag propagation, which is central to information flow theory and dynamic information flow tracking.

After the filtering phase, we utilized the pool of relevant works to once more search for additional relevant works, by manually going through the references of each already designated related work and the works that cited our already designated related works. This process left us with 50 designated relevant works, which were then submitted to the final phase, *categorization*.

3.4 Categorization and Sorting

The final phase of our systematic literature review was to compile data from each relevant work. The initial step was, for each relevant work, conduct an in-depth reading of the manuscript to determine the most relevant categories across all relevant works. A few examples of questions used in this analysis are: "Software or Hardware Focus?", "Tag Format", "Tag Granularity", "Whole System or Application Specific", "Tracks Indirect Flows?", and "Collaboration between Hardware and Software?".

Table 1 includes a distilled set of the key properties and characteristics of all collected references (excluding those that were filtered out), which were used to sort each relevant work into its respective category. Across all of the references chosen, four major categories of papers emerged: (i) *DIFT Fundamentals*, containing works related to fundamental attributes of DIFT systems, such as tag size, format, and how indirect flows are handled, (ii) *DIFT in Software (SW-DIFT)*, capturing works implementing DIFT purely in software, (iii) *DIFT with Software-Hardware Collaboration (SW-HW DIFT)*, containing works implementing DIFT by leveraging both the software and hardware subsystems, and (iv) *DIFT in Hardware (HW-DIFT)*, covering works implementing DIFT purely in hardware. These categories were developed iteratively and unanimously agreed upon the researchers/coders throughout careful paper reviews.

Figure 1 shows the distribution of DIFT categories over the years. Across different implementations of DIFT, we found that $\tilde{6}7\%$ of the research works on DIFT focused on enforcement of security policies – $\tilde{3}3\%$ on confidentiality, $\tilde{2}9\%$ on integrity, and $\tilde{5}\%$ on both. Although, a small fraction of works have employed DIFT for both confidentiality and integrity, they did not investigate the scenario when these two policy models (one the dual of the other) were applied at the same time. Approximately one third of the reviewed works leveraged DIFT for non-security purposes, such as program visualization [51, 54]).

Table 1. Reference Properties of the DIFT works reviewed in this paper.

Works	Theme	Tag Format	Tag Granularity	System or App Specific	Tracks Indirect Flows	Indirect Flow Method	Run-time Increase
She et al. [67]	DIFT Fund.	1-bit or 8-bit	1-byte	Whole System	Addr. Dep. Only	Unspecified	-4000% Libdft
Santos et al. [50]	DIFT Fund.	Split memory[1]	N/A[1]	Whole System	Yes	No	Negligible
Suh et al. [71]	DIFT Fund.	1-bit	byte/word/page	Whole System	No	N/A	1.1% to 23%
Sapountzis et al. [64]	DIFT Fund.	Provenance List	1-byte	App Specific	Yes, variably.	Amnt. info added	Unspecified
Sapountzis et al. [65]	DIFT Fund.	Provenance List	1-byte	Whole System	Yes	Parameterized Eqn.	3360%
Al Saleh et al. [1]	DIFT Fund.	8-bit fixed point	1-byte	App Specific	Yes	Tunable tag	Unspecified
Slowinska et al. [69]	DIFT Fund.	32-bit	1-byte	Whole System	Yes	No	N/A
Schwartz et al. [66]	DIFT Fund.	N/A[2]	N/A[2]	N/A[2]	N/A[2]	N/A	N/A
Ji et al. [38]	DIFT SW	1,8, or more bits	1-byte	Whole System	No	N/A	4.84%
Egele et al. [28]	DIFT SW	1-bit	1-byte	Whole System	No	N/A	300%
Mazloom et al. [51]	DIFT SW	4-bytes	1-byte	Whole System	No	N/A	Avg: 10000%
Mysore et al. [54]	DIFT SW	4-bytes	1-byte	Whole System	No	N/A	0% to 100% avg
Dolan-Gavitt et al. [27]	DIFT SW	Arbitrary	Arbitrary	Whole System	Configurable	App. Specific	App. Specific
Kang et al. [40]	DIFT SW	1-bit	1-byte	Whole System	Yes	Static Analysis	App. Specific
Espinoza et al. [30]	DIFT SW	Vec. of 200 floats	1-byte	App Specific	Yes	Tag weights	N/A
Newsome et al. [56]	DIFT SW	1-bit	1-byte	App Specific	No	N/A	200%-3720%
Arefi et al. [4]	DIFT SW	Provenance List	1-byte	Whole System	No	N/A	5600%
Yin et al. [83]	DIFT SW	Data Struct.	1-byte	Whole System	No	N/A	2000% slowdown
Clause et al. [18]	DIFT SW	Bit-vector	1-byte	App Specific	Yes	No	3000%
Chen et al. [15]	DIFT SW	Configurable	Configurable	App Specific	Yes	No	5%-25%
Kemerlis et al. [42]	DIFT SW	1-bit or 8-byte	1-byte	App Specific	Addr. Dep. Only	No	114% to 603%
Ji et al. [37]	DIFT SW	1-bit or 8-bit	1-byte	Whole System	No	N/A	4.84%
Ozsoy et al. [57]	DIFT SW	1-bit or 1-byte	Per-Address	Whole System	No	N/A	26%
Qin et al. [62]	DIFT SW	1-bite	1-byte	Whole System	No	N/A	Unspecified
Chang et al. [12]	DIFT SW	32-bit	1-byte	App Specific	No	N/A	0.65%
Banerjee et al. [6]	DIFT SW	Bit-vector	1-byte	App Specific	No	N/A	1.07%-1.12%
Zhu et al. [85]	DIFT SW	1-bit	1-byte	App Specific	No	N/A	Unspecified
Costa et al. [19]	DIFT SW	1-bit	Per page	Whole System	No	N/A	<1%
Townley et al. [76]	DIFT SW-HW	1-word	Per page	Whole System	No	N/A	<1% to 60%
Dalton et al. [23]*	DIFT SW-HW	4-bit	Per word	Whole System	No	N/A	1% to 3000%
Porquet et al. [61]	DIFT SW-HW	Bit-vector	Per word	Whole System	Unspecified	N/A	22% to 50%
Vachharajani et al. [78]	DIFT SW-HW	N/A[3]	N/A[3]	Whole System	Yes	ISA + Bin. xlation	0% to 100%
Chow et al. [17]	DIFT HW	1-bit	1-byte	Whole System	No	N/A	N/A
Kannan et al. [41]*	DIFT HW	4-bit	32-bit	Whole System	No	N/A	1%
Lee et al. [47]*	DIFT HW	1-bit	Per word	App Specific	No	N/A	1.6%
Santos et al. [49]	DIFT HW	1-bit	1-byte	App Specific	Unspecified	N/A	10%
Crandall et al. [21, 22]	DIFT HW	1-bit	Per word	Whole System	No	N/A	Negligible
Pilato et al. [60]*	DIFT HW	1-bit	Configurable	App Specific	No	N/A	0% to 100%
Joannou et al. [39]*	DIFT HW	1-bit	64-bit	Whole System	No	N/A	1%
Wahab et al. [79]*	DIFT HW	1-bit to 32-bit	Configurable	App Specific	Unspecified	N/A	335%
Wahab et al. [80]*	DIFT HW	1-bit	Configurable	Whole System	Yes	ISA + Static Analysis	5.35% to 24.6%
Palmiero et al. [58]*	DIFT HW	1-bit	1-byte	Whole System	Yes	No	Negligible
Chen et al. [13]*	DIFT HW	1-bit	1-byte	Whole System	No	N/A	Negligible
Piccolboni et al. [59]*	DIFT HW	1-word	1-byte	Whole System	Unspecified	N/A	0% to 100%
Tiwari et al. [75]*	DIFT HW	1-bit	1-bit	Whole System	Yes	ISA Design	Unspecified
Bosman et al. [10]	DIFT HW	1-bit	1-byte	Whole System	No	N/A	150% to 300%
Tiwari et al. [74]*	DIFT HW	1-bit	1-bit	Whole System	Unspecified	N/A	Unspecified

* represents the work was evaluated on actual hardware platform, e.g., FPGA prototype.

[1] These are N/A, as PIFT [50] splits tainted and untainted data into designation blocks of memory.

[2] These are N/A, as this work is an older survey of DIFT.

[3] These are N/A, as RIFLE [78] translates existing instructions into information flow security instructions that propagate and store taint information equivalent to the existing size and format of data in memory.

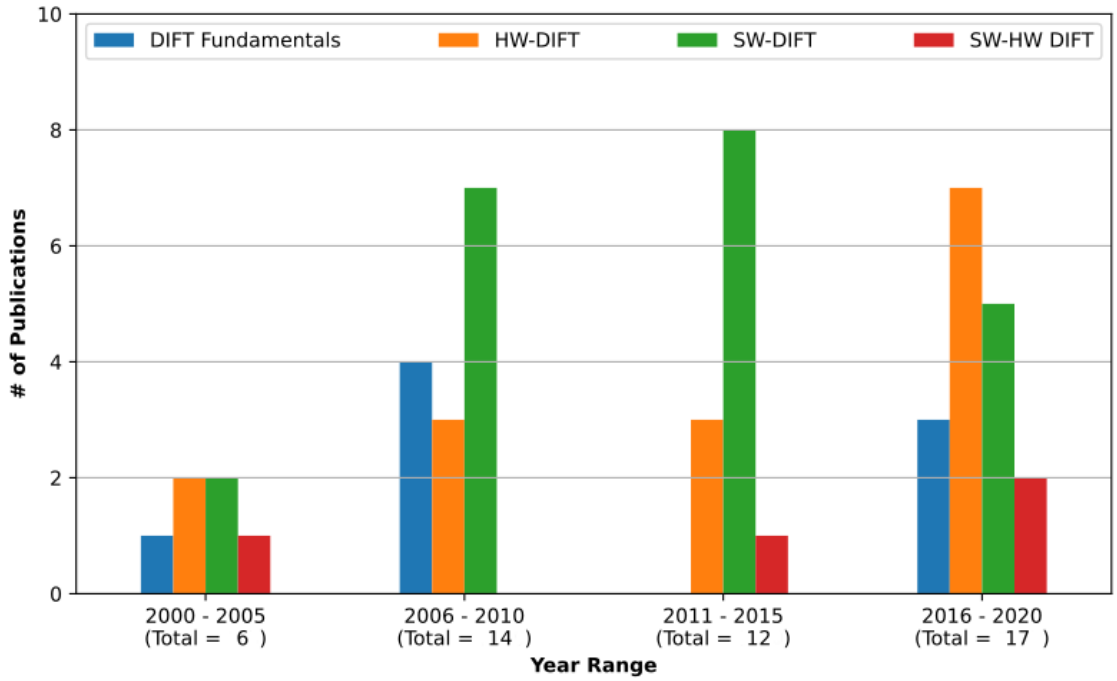

Fig. 1. Category distribution of DIFT literature over the years.

4 DIFT FUNDAMENTALS

ADDRESSING RQ1: WHAT ARE THE KEY ATTRIBUTES OF A DIFT SYSTEM THAT INFLUENCE ITS PERFORMANCE AND ACCURACY?

In DIFT, certain inputs are tainted with metadata *tags* (*Tag Insertion*), which are propagated as the program/system runs based on a given policy (*Tag Propagation*). These tags are subsequently checked for program analysis and/or security policy enforcement (*Tag Checking*), as described below.

4.1 Tag Basics

Size and Format: The research community has experimented with a diverse range of tag (sometimes denoted as "taint" in the literature) sizes and formats. Tags may be a single bit or multiple bytes, and even data structures, depending on the layer of abstraction of the DIFT solution and the security policy to be enforced. For example, Minos [21, 22] a microarchitecture for integrity enforcement, Bitblaze [70], a whole-system DIFT architecture for binary analysis, and the whole-system proposed by Suh et al. [71] are the earliest examples of DIFT variants that utilize single bit tags. While the use of single bit tags can achieve good accuracy in detecting simple policy violations (e.g., buffer overflows) with reasonable memory and runtime overheads, its simplistic design lacks expressiveness. For instance, a system using 1-bit tags cannot differentiate between a tag inserted at the network interface and one inserted via a system call invocation. The inability to differentiate tags limits the information that can be collected and utilized for more rigorous analysis.

To address this limitation, thus providing higher expressive power compared to that of one-bit schemes, subsequent DIFT systems, such as Dytan [18], WHISK [61], and that by Kannan et al. [41], proposed tags of multiple bits (or variable bit sizes). Dytan utilizes variable length bit vectors, so that security analysts can define if they want to taint data from specific sources. WHISK uses variable length bit-vectors as well, however the authors define the options for the lengths to only be 1, 8, 16, or 32 bits to observe the degree to which tag bits are utilized with regards to the tag width. Kannan et al. [41] employed 4-bit tags per word in memory, such that low-level memory corruption attacks and high-level semantic attacks can be detected.

Crucially, these multi-bit schemes [18, 41, 61] allow for policies that can differentiate the sources of tags. Examples of such a highly expressive DIFT implementation include those leveraging tags of provenance list style, such as FAROS [4] and DDIFT [64]. FAROS is a whole-system DIFT approach for reverse engineering running on top of PANDA/QEMU, in which provenance lists contain sets of tags of different types (e.g., network information, file information), allowing for the identification of tag signatures of in-memory-only attacks. DDIFT is a DIFT framework for IoT devices that leverages tracking at the device level and optimization of tracking at the cloud level, and similarly to FAROS, utilizes provenance information for attack detection. These systems can track information about the history of tagged data within the system, e.g., where the data has been, or what other data it has interacted with. However, the increased expressiveness of such a multi-bit scheme comes at the high cost of increased memory and runtime overheads. Moreover, the larger and the more complex a tag is, the more challenging it is to have the DIFT system implemented in hardware.

Tag Granularity: Tag granularity is a measure of the smallest-sized memory unit that a tag can be associated with. It determines three critical performance metrics for a DIFT system – memory overhead, runtime overhead, and information accuracy. For a DIFT scheme with a simple tagging policy (e.g., 1-bit tags), the benefits of fine tag granularity (e.g., 1-byte granularity), such as the ability to detect buffer overflow attacks, can outweigh the storage overhead costs. Works such as libdft [42] and that of Chow et al. [17] justify their choice on byte level granularity as improving accuracy in flagging policy violations or keeping the DIFT architecture simple.

While a DIFT scheme with coarse-grained tag granularity (e.g., word level or higher) can reduce memory overhead, it can potentially leave some data unnecessarily tainted (false positive) [17]. To combine the best of two worlds, Suh et al. [71] proposed multi-granularity security tags, where the operating system (OS) maintains two extra tag bits per page to indicate the granularity within that page, which allows the system to reduce the amount of shadow memory allocated for tag storage (discussed below). The cost of this approach is that it requires the shadow memory to be dynamically allocated by the DIFT-aware OS. However, we believe that moving forward, multi-granularity schemes are likely to become a general standard as the field progresses, as more complex implementations start exploring more the principles of spatial and temporal locality of a taint.

4.2 Tag Insertion

Tag insertion is the process of inserting new *tags* into a running system. Tags are usually associated with memory units involved in certain system activities and can be inserted into (or associated with) policy-defined regions of memory. Tags can be inserted at any point designated by a chosen security policy. In most cases, tags are inserted at points in a system where data is being used in an untrusted way or entering the system from an untrusted source. In other cases, data that flows through designated I/O channels will be tagged, or any untrusted application or process may have its data tagged. For example, TaintBochs [17], which utilizes 1-bit tags, considers all data that enters their system from the network as tainted (untrusted) and tags them immediately upon their entering in the system.

Additionally, some systems initialize their data with certain taint values, which are then propagated during execution, so that an analyst can observe which areas of memory most interacted with each other. An example of this is V-DIFT [30], which inserts a vector of random floating points values as the tag for every byte in its system for locating the region of memory where a cryptographic key is stored by different cryptographic algorithms. The policy used by V-DIFT dictated tag insertion at the beginning of execution. Other systems may insert tags at certain interfaces, e.g., when data enters the system via user input, upon invocation of a system or function call (e.g., FAROS [4]), or via a network interface (e.g., MINOS [21, 22]).

4.3 Tag Storage

The original works by Denning and Fenton [25, 26, 31, 32] do not address the logistics associated with storing tags, yet tag storage sustains as an integral aspect of a functioning DIFT system. The tags employed by a DIFT system must be stored such that they are accessible for propagation and policy checking. The location of tag storage varies for each individual DIFT scheme. Generally, the memory that DIFT systems use to store tags is denoted as *shadow memory*.

DIFT implemented entirely in software often has its shadow memory located within the main memory, like Pagurus [59], FAROS [4], and DDIFT [64]. These DIFT systems store tags in a segment of main memory such that addressing and accessing stored tags can be mapped in memory in a flexible way. DIFT implemented in hardware, or in software with hardware modifications, typically stores tags in its own physical shadow memory. This can be implemented as a fully separate memory module or as adjacent memory [21, 22]. Minos, implementing DIFT in an x86 emulator, utilizes adjacent shadow memory by simply extending each unit of main memory by the length of the tag (1 bit). Similarly, Pagurus has a configurable mode that allows for adjacent storage of tags, called a "coupled" scheme. In other instances, tags are stored within the space of the program (or the application), one example being PIFT [50], which taints memory at page granularity and allocates memory at compilation time into *trusted* or *untrusted* regions, with the goal of achieving zero memory overhead in their system.

There are benefits and detriments to any chosen shadow memory scheme. Shadow memory implemented as a separate segment, either in software or hardware, facilitates superior flexibility of mapping storage, modularity, and allows for isolation of tag storage. On the other hand, accessing segmented shadow memory is not as fast as accessing adjacent shadow memory. However, adjacent shadow memory is far less flexible, as it cannot be mapped or utilized in any fashion different than the original design. More unique DIFT implementations, like PIFT [50], achieve zero memory overhead in their system as all data is stored in variable trusted or untrusted regions of memory. Unfortunately, this creates performance overheads from the increase in memory accesses to ensure all data is moved into its correct and respective region of memory.

4.4 Tag Propagation

To regulate the flow of information within the program (or the system), the inserted tags are propagated as the program or system runs according to the information flow policies in place (e.g., Biba for integrity). There are two types of tag propagation methods, as described below.

4.4.1 Direct Flow Propagation. Direct flows are comprised of copy and computation dependencies. In a copy dependency, a value is copied from one location (e.g., a byte or word in memory or CPU register) to another. Comparatively, in a computation dependency, a function is applied to data values. To track the information flow of a copy dependency, the tag is propagated from the source tag location to the destination tag location. In the case of computation dependency,

the tags of the values used for calculation are combined as a union. For example, when the computation of the sum between two variables results in a third variable, e.g., $z = x + y$, the tag of the resultant variable (i.e., z) contains the union of the tags of the two variables x and y.

4.4.2 Indirect Flow Propagation. Indirect flows are based on the control-flow of a system. In particular, *indirect* flows are a property of *Dynamic* Information Flow Tracking, whereas *implicit* flows, as described by Denning [25]. are a property of static analysis of information flow. Specifically, an indirect flow is an occurrence of an implicit flow within dynamic analysis, as described by V-DIFT [30]. The designation of implicit flow entails the knowledge of the different possible execution paths, whereas indirect flows only have knowledge of the execution path that has been observed to have executed.

Indirect flows can be divided into two types – *address* and *control* dependencies. Address dependency occurs when a resultant value is dependent specifically on the initial value of a certain memory region, which is used to address (or index) another value. Specifically, Figure 2 [4] presents an example of address dependency in C that utilizes a lookup table to propagate a tainted string from one location to another. Since the input string `oldString` is tainted, the resultant string `newString` should also be tainted. In the example, to ensure that `newString` has been tainted properly, the taint status of the address (used for the load) is checked with `lookupTable` and this taint is propagated. In a control dependency, the value within a certain memory region determines the flow of the program instructions. Figure 3 [4] shows an example of control dependency in C, where the current value of variable `bit` determines the value of variable `untaintedByte`.

```
const char oldString = "TaintedString";
char newString[32];
char lookupTable[256];
for (i = 0; i < 256; i++)
    lookupTable[i] = i;
for (j = 0; j < strlen(oldString); j++)
    newString[j] = lookupTable[oldString[j]];
```

Fig. 2. An example of address dependency in C [4].

```
char taintedByte;
char untaintedByte = 0;
for (int bit = 1; bit < 256; bit <<= 1) {
    if (bit & taintedByte)
        untaintedByte |= bit;
}
```

Fig. 3. An example of control dependency in C [4].

Undertainting and Overtainting:

One of the major challenges of making DIFT a practical security technology is that while it is straightforward to observe occurrences of direct flows, indirect flows are significantly harder to track. Therefore, current DIFT systems lack

the ability to discern the necessity of propagating tags through indirect flows, revealing the dilemma of undertainting vs overtainting. Overtainting occurs when tags are propagated through all indirect flows, leading to taint explosion. Slowinska and Bos [69] define *taint explosion* as a phenomenon in which a tag spreads rapidly within a system during execution. This results in *taint pollution*, as most objects in the system are tainted, leaving the memory saturated with tags and no longer able to produce meaningful results in terms of information flow tracking. In other words, overtainting can make it impossible to discern the actual behavior of a system. The challenge of tracking indirect flows and the performance costs of overtainting leads to nearly all DIFT solutions to only track direct flows. Conversely, undertainting occurs when tags are not propagated enough through indirect flows, leading to the loss of pertinent knowledge about the information flow (low accuracy). The information that has been lost due to undertainting may be crucial for security applications to detect/prevent the attack and the violation of security policies (see examples in the discussion below).

Few works [30, 65, 69] have attempted to discuss or address the undertainting vs. overtainting dilemma. Besides Slowinska and Bos [69], V-DIFT [30], an application-specific DIFT method for locating cryptographic keys within memory, attempts to address the issue of tracking indirect flows by utilizing vectors as taint marks and weighting the propagation of address or control dependencies to decrease the likelihood of taint explosion. MITOS [65] analytically models the problem of optimizing the propagation of indirect flows and discovered two properties to be key factors for the efficiency and usefulness of modern DIFT systems: fairness and tag-balancing. Fairness is valuable to properly balance tag propagation, based on the pervasiveness of certain types of tags in the system. In other words, the DIFT system would reduce the likelihood of pervasive tags being overly propagated. While MITOS introduces new insights into creating an optimal solution, this dilemma is still open and constitutes one of the major refraining factors to real-world implementations of DIFT systems.

We note that assumptions that integrity-focused applications do not need to reason about indirect flows are flawed. For example, Crandall and Chong [21] showed that many control-flow hijaking attacks they analyzed were not detected by DIFT without tracking indirect flows. Conversions between formats, compression and decompression, encoding and decoding, and many other common operations apply indirect flows of information to the data they operate on, and every pointer calculation (such as for an array) and conditional branch (such as if statements and for loops) are opportunities for indirect flows. Thus, the complete and correct tracking of indirect flows are key for the development of practical and effective security-based DIFT solutions.

4.5 Tag Checking

The purpose of tag checking is to verify whether information is flowing according to a defined security policy. Specifically, DIFT checks if the current flow of data to or from one object caused by a subject action follows a pre-defined security policy and ensures that the policy is enforced. For example, Minos enforces the Biba integrity policy [21, 22] to prevent control-flow hijacking attacks, such as buffer overflows. Minos employs a single “integrity bit" tag to indicate if its corresponding data is low integrity (tainted). Minos taints all bytes coming from the network and considers a security violation when tainted data is written into the PC register. For a real-world implementation of DIFT, the consensus is that tag checking must occur regularly enough (e.g., for every instruction executed), such that all violations to the security policy are detected.

Checking tags too often, however, can incur in performance overheads. LATCH [76], for instance, utilizes software-hardware collaboration to enforce whole-system integrity and attempts to decrease the time to process tags by checking them only when necessary, based on spatial-temporal locality of the tags within the memory system, by only initiating

fine-grained taint checking and propagation when the system detects tags at the page level. LATCH checks operand tags using a coarse-grained taint checking hardware module, and in the event a tag is detected at the course granularity, it then temporarily resorts to byte granularity for propagation and checking. Due to the strong temporal locality exhibited by the tags, this process allows for the resource intensive monitoring to be invoked less often. There are also a few works that do not use explicit rules to determine tag propagation and checking, such as that by She et al. [67].

4.6 Summary

Thus far, DIFT systems can vary many key attributes to implement taint tracking functionality, such as tag size, format, and granularity. Tag insertion can be handled when the system is initiated or via an event (e.g., a system call invocation or the arrival of a network packet), and tags are stored in segments of main memory or in dedicated shadow memory modules. Tags are propagated as the program or system runs, based on direct or indirect flows. During propagation, the creation or transfer of tags to objects are checked against static or configurable rules to enforce a desired security policy. The design choice for the values of the different attributes of a DIFT system (tag size, granularity and frequency of tag checking) has a direct impact on the system's performance, memory usage, and accuracy in exposing important information flows. For example, larger and more complex tags can improve accuracy, in that more information about an object can be stored and propagated. However, larger tags require more shadow memory space and more processing time for tag checking. The same rationale is valid for applying tags at a finer granularity, e.g., tagging every byte vs. tagging a word or an object: tagging at a finer abstraction level leads to more tags coexisting in the system, which requires more memory space and incurs in higher tag processing time. On the other hand, tagging data at a finer granularity improves system accuracy by preventing, for example, a potential malicious byte evading exposure because it was group-tagged with a piece of data coming from a trusted source. Similarly, the cadence of tag checking also affects system performance and accuracy: frequent tag checking incurs more CPU cycles, but increases the likelihood of a malicious flow of information being detected. An open research challenge is how to perform tag insertion and checking in an optimal fashion, without falling for the overtainting vs undertainting dilemma, which directly affects the performance and the accuracy of the DIFT system. Even though recent research suggested promising optimization heuristics for determining when to tag [38, 64], such implementation in hardware is still seen as unfeasible due to added complexities required for the hardware subsystem.

5 DIFT IN SOFTWARE (SW-DIFT)

ADDRESSING RQ2: WHAT IS THE STATE OF THE ART FOR DIFT SOLUTIONS IMPLEMENTED EXCLUSIVELY IN SOFTWARE?

Most of the DIFT proposals in the literature are software-based, referred to in this paper as *SW-DIFT*, potentially due to the flexibility software offers to implement propagation rules and configure parameters, such as tag format, size, format, granularity, and storage mechanism. Such flexibility streamlines the development of various DIFT applications, such as security policy enforcement, visualization, reverse engineering, and malware analysis.

SW-DIFT needs to introduce DIFT-based machine-level instructions into the program binary, which can be accomplished at the source code level or via binary translation. For the introduction of DIFT operations in the source code, a special compiler is needed to insert the respective machine-level instructions in the binary program. One challenge is how to enforce policies in third party libraries or proprietary and commodity programs, where the source code is not available. Furthermore, given today's software supply chain high dependence on externally written open-source software (packages, libraries, or modules), the need for compiler modifications is a significant limitation [72, 73].

On the other hand, while applying DIFT operations into executable code via binary translation (re-writing) [16, 18, 56, 62] can capture third-party library/proprietary code, the large performance and memory overhead costs of this approach challenge their on-the-fly applicability [37]. For RAIN [37] the memory overhead incurred was typically around 50%. Specifically, in whole-system analyses, binary translation DIFT becomes heavy-weight tracking, causing prohibitive slowdowns (e.g., FAROS's 56x runtime overhead [4]). Moreover, binary translation approaches face challenges while handling multi-threaded programs due to the complexity of control and data dependency propagation among threads.

While there is a wide variety of SW-DIFT proposals, they can be categorized into two classes – i) *Online SW-DIFT*, applying DIFT on-the-fly during the program execution and ii) *Offline SW-DIFT*, applying DIFT without performance concerns and/or via record-and-replay of the program or the whole system.

5.1 Online SW-DIFT

This category of DIFT system provides run-time and flexible detection mechanism, but can generate substantial performance overhead, especially if the scope of taint-tracking operations is the whole system and not a single application. Online analysis implementations have also been leveraged for non-security applications, such as program visualizations [42, 51, 54]. Such visualizations are used to create diagrams and form an understanding of the code, or to detect misbehavior during program execution. In the case of the DIFT systems proposed by Mazloom et al. [51], the slowdown was minimal for compression operations, but for database operations, the average slowdown was 100x, and up to 200x for deleting database entries. Representative examples of online SW-DIFT systems proposed in the literature are Vigilante [19], TaintCheck [56], Dytan [18], libdft [42], and TaintDroid [29], summarized below.

Vigilante [19], one of the first online whole-system SW-DIFT, targeted Internet worm detection. The system is basically an x86 application performing binary re-writing at load time and tagging data at the byte-level. In Vigilante, every control transfer instruction (e.g., RET, CALL, JMP) and every critical and data movement instruction are instrumented with tags stored in main memory. Proposed concurrently, TaintCheck [56] is an online whole-system fine-grained SW-DIFT solution performing binary translation at runtime with the goal of detecting buffer overflow attacks and generating signatures against detected exploits.

Clause et al. [18] introduced Dytan, an application-specific online SW-DIFT based on binary translation. Unlike Vigilante and TaintCheck, Dytan attempts to address control dependencies (indirect flows). It utilizes a conservative approach to track control dependencies by computing and propagating a tag for any result of a conditional branch statement. This approach is directly inspired by that of Vachharajani et al. [78] (see Section 6.1 for a discussion). Dytan also allows the DIFT analyst to define and configure three core dimensions of DIFT, namely the taint source, i.e., the memory location that should be tagged, the propagation policy, and the taint sink, i.e., the location for tag checking. More importantly, it allows a security analyst to define tag propagation rules in the form of a mapping function, the default behavior being to taint the produced data with the union of tags of all tainted operands.

As a follow-up to TaintCheck, Kang et al. [40] introduced DTA++, a whole-system online SW-DIFT, to track implicit flows accurately by propagating tainted control dependencies selectively. Specifically, TaintCheck propagates a tag along a targeted subset of control-flow dependencies to protect the confidentiality of sensitive data. It first detects indirect flows that lead to undertainting, then adds additional tags only for those control dependencies at a byte-level granularity. The goal is to ameliorate the undertainting vs. overtainting dilemma.

Kemerlis et al. [42] introduced libdft, an application-specific online SW-DIFT framework for commodity systems, that provides an API to support DIFT-enabled tools for unmodified binaries. Specifically, it is a shared library that implements DIFT using Intel's Pin dynamic binary instrumentation framework. There are three categories that a libdft-enabled tool

can use as a data source or sink: program instructions, function calls, and system calls. libdft stores tags in a tagmap containing a process-wide data structure and a thread-specific structure. The process-wide data structure holds the tags for data in memory, while the thread-specific data structure holds tags for data in registers. Although, libdft can track indirect flows for address dependencies, it does not consider control dependencies.

Enck et al. [29] proposed TaintDroid, the first whole-system DIFT for Android smartphones, to track the flow of privacy-sensitive data through third-party applications. TaintDroid monitors the access and manipulation of users' personal data by third-party applications in real-time. Leveraging Android's virtualized execution environment, TaintDroid integrates four granularities of tag propagation – variable-level, method-level, message-level, and file-level. While TaintDroid incurs less than 32% of performance overhead on CPU-bound microbenchmarks and negligible overheads on interactive third-party applications, it can be circumvented through leaks via indirect flows.

5.2 Offline SW-DIFT

Several SW-DIFT systems for offline analysis are based on the *record-and-replay* technique, which records all the events and states during program execution and replays the recorded events while applying taint tracking operations, usually for forensic purposes [38]. Given its offline nature, this approach provides the flexibility to examine a system in detail without performance overhead constraints. Offline SW-DIFT plays a vital role in reverse engineering applications and employs different techniques, including the tracing of the point when the malware uses code unpacking [28], or conducting malware behavior analyses by monitoring system or API function calls [4].

For instance, Panorama [83] is a whole-system fine-grained SW-DIFT mechanism intended for automatic malware detection. It is implemented on QEMU, a generic processor emulator, and aims to protect the confidentiality of data by exposing privacy-breaching malware. The main idea is to monitor the behavior of the sample program with respect to the sensitive information and log relevant information in the form of a taint-graph (byte level tags). DIFT (only direct flows) is leveraged to track the sensitive information propagation within the whole system and its propagation into the sample program (i.e., whether the program has exfiltrated the information). To detect malware, Panorama checks for a violation of pre-defined confidentiality policies on the taint graph.

PANDA [27] was developed as a tool to facilitate offline analysis (reverse engineering) of a whole system. It stores tags in a shadow memory and the tag propagation is handled by inline LLVM code. PANDA employs QEMU virtualization and the record-and-replay technique for deep vulnerability analysis. FAROS [4] extended PANDA to track and detect in-memory-injection attacks in Windows systems using provenance lists as tag granularity.

V-DIFT [30] is an application-specific SW-DIFT solution intended for reverse engineering of malware that attempts to discover cryptographic keys in memory. It uses vectors to represent tags to streamline tag processing via vector operations and tag comparison via cosine similarity. The use of vectors provides an approximation of the information flow considering indirect flows, despite the approach not completely solving the overtainting vs. undertainting dilemma.

RAIN [37] is a whole-system SW-DIFT that leverages its custom record-and-replay mechanism to construct a logical provenance graph by continuously monitoring and logging all the system-call events. The coarse provenance graph is compressed and sent to a trusted host from the target host for further analysis. RTAG [38] extended RAIN's record-and-replay technique with a custom tagging system to facilitate investigation of cross-host attacks. Specifically, it utilizes the reachability analysis of RAIN (i.e., pruning the system-wide provenance graph to extract a subgraph related to the designated attack) and extends it to cope with cross-host scenarios.

5.3 Summary

The greatest advantage of SW-DIFT is the flexibility in implementation, which allows researchers to build applications for a variety of goals and systems. Implementation of DIFT in software is accomplished by either specialized compilers adding special instructions to perform taint tracking into the software source code or via binary instrumentation. For SW-DIFT relying on source-code transformations, the challenge is extending taint-tracking operations to third-party libraries and proprietary code. While binary translation can extend taint tracking to third-party code, it incurs significant performance overheads, which limits its applicability for on-the-fly operation. That is why many SW-DIFT proposals have a natural offline/reverse engineering applicability for malware/intrusion analysis.

While some proposals attempted to address indirect flows [4, 18, 69], most still focus on only tracking direct flows. Moving forward, our expectation is that SW-DIFT will continue to be a fertile ground for prototyping different implementations and applications and as analysis/reverse engineering tools to be used offline. However, practical uses will move to implementations in hardware or relying on hardware-software collaboration.

6 SOFTWARE-HARDWARE COLLABORATION FOR DIFT (SW-HW DIFT)

ADDRESSING RQ3: WHAT IS THE STATE OF THE ART FOR DIFT SOLUTIONS IMPLEMENTED BY LEVERAGING BOTH THE SOFTWARE AND HARDWARE SUBSYSTEMS?

The literature has given evidence that there is much promise in combining the flexibility a software implementation provides with the performance gains from executing taint tracking operations in hardware for real-time operation, resulting in SW-HW DIFT systems, which are typically implemented as i) *Hardware-Supported Software* or ii) *Software-Supported Hardware*, as described below. While such systems have positive improvements on performance and memory overhead, they require substantial modifications in both hardware and software subsystems.

6.1 Hardware-Supported Software

We categorize a SW-HW DIFT system as leveraging hardware-supported software if it primarily comprises of a customized software system (or a modified ISA) to handle DIFT operations and a modified hardware (processor) platform to facilitate the implementation of the ISA. The modification of the conventional ISA allows the DIFT system to achieve flexibility in implementation choices.

One instance of such design is RIFLE [78], a run-time information flow security system, that aims to protect the confidentiality of data by enforcing user-defined security policies. It is based on an abstract information flow architecture (Information Flow Security -IFS- ISA) that translates a targeted program binary into a modified binary for the custom IFS ISA. The translated binary is executed within a modified processor supporting the IFS ISA, which is simulated within the Liberty Simulation Environment [77]. The simulated processor interacts with a security-enhanced OS to enforce the user-defined policies. RIFLE also highlights the impracticality of Fenton's data mark machine [31, 32], as the compiler requires knowledge of the privilege of every piece of data at every execution point. Therefore, for practical purposes, Vachharajani et al. [78] suggest that the compiler could instead enforce security statically with Denning's lattice model [25, 26].

6.2 Software-Supported Hardware

We classify a SW-HW DIFT system as software-supported hardware if the hardware component provides an architectural structure for fundamental DIFT operations, such as tag storage, propagation, and checking, and the software component

supplements the architecture by handling more complex DIFT operations, such as security exceptions, tag storage allocation, and precise tag checking. This model allows hardware-based DIFT implementations to gain functional flexibility from the software support at the expense of increasing their dependence on complex software modifications to support the specialized hardware.

One example of such design is Raksha [23], an architecture to assure the implementation of DIFT-based integrity policies. Raksha is implemented as an FPGA prototype, which integrates the hardware support for tag propagation and analysis with additional software analysis to support flexible and programmable security policies that can provide protection against high-level (e.g., SQL injection) and low-level (e.g., buffer overflows) attacks. Raksha delegates tag storage, access, and propagation to the hardware, while implementing security handlers and taint analysis in software within a custom Linux distribution.

Similarly, WHISK [61], a whole-system DIFT architecture implemented within a hardware simulator, consists of: i) hardware architectural mechanisms to integrate DIFT in heterogeneous System on Chip (SoC) designs and ii) software wrappers that provide DIFT functionality for third-party IP components. WHISK stores tags and data separately in memory locations to keep low area overhead and improve flexibility. It delegates tag insertion, storage, and access to the custom hardware. The software subsystem, which utilizes the MutekH exokernel-based OS [2], provides support for tag page table allocation, page table cache configuration, and interrupt handling concerning writes to untagged pages. WHISK's goal is to facilitate integration of DIFT in SoCs, without a focus on individual security policies. Such a hardware-software collaboration allows the system to maintain expressiveness, while reducing performance overhead (7.5%).

LATCH [76] leverages the notion that information flows have a high degree of spatial and temporal locality to minimize the time consumed during precise taint checking, which is key to its effectiveness. The main goal is to detect system integrity breaches. The architecture consists of a software-supported hardware accelerator (S-LATCH) running on a single simulated core. S-LATCH's software component propagates tags, while the hardware accelerator monitors the data accessed by the program to detect tags. When a tag is detected, the hardware accelerator invokes the instrumented OS via a hardware exception for policy enforcement at an average of 60% runtime overhead.

6.3 Summary

Software and hardware collaboration to support DIFT has been accomplished in two ways: hardware-supported software and software-supported hardware. These approaches typically aim to implement DIFT with programmable or variable policy enforcement. The performance overhead of such design is lower than its purely software counterpart. A popular approach has been to introduce a custom ISA running on specialized hardware. This architecture facilitates policy programmability via software support or utilizes a hardware accelerator that triggers when to apply software support. However, while these approaches attempt to combine the best software and hardware have to offer (programmability and good performance), they introduce the challenge of requiring substantial and non-trivial modifications both in software and in hardware (customized OS, ISA and hardware), which might discourage practical, widespread applications of this design.

7 DIFT IN HARDWARE (HW-DIFT)

ADDRESSING RQ4: WHAT IS THE STATE OF THE ART FOR DIFT SOLUTIONS IMPLEMENTED EXCLUSIVELY IN HARDWARE?

DIFT designs implemented in hardware, referred to as *HW-DIFT*, can be classified into three categories – (i) *in-core*, where the whole processor core is modified to support DIFT operations [23, 58], (ii) *off-core*, where all DIFT operations

are performed outside the processor core [13, 46, 47, 80], and (iii) *off-load*, where a separate core is dedicated specifically for the DIFT operations [55].

Table 2. Hardware Based Information Flow Tracking Techniques.

Works	Type	Performance Overhead	Hardware Overhead
Chow et al. [17]	In-Core	Unspecified	Unspecified
Minos (Crandall and Chong [21])	In-Core	+3.125%	Unspecified
Palmiero et al. [58]	In-Core	Unspecified	+12.5%(storage),+0.82%(LUT)
Tiwari et al. [74])	In-Core	-32%	+5% to +35%(ALUT)
Tiwari et al. [75]	In-Core	Unspecified	+70%(ALUT)
Raksha (Dalton et al. [23])	In-Core	+34%	+12.5%(Storage)
Kannan et al. [41]	Off-Core	+1%	+8%
Lee et al. [46]	Off-Core	-82.9%	+60%(BRAMs), +27.98%(LUTs)
Lee et al. [47]	Off-Core	-82.9%	+60%(BRAMs), +28.36%(LUTs)
Wahab et al. [79]	Off-Core	-26%	+1%(Area), +16.2%(Power)
Wahab et al. [80]	Off-Core	-5.4%	+5.87%(AXI), +5.2%(IP), +4.62%(Softcore)
Chen et al. [14]	Off-Load	+2% to 51%	Unspecified
Nagarajan et al. [55]	Off-Load	+48%	Unspecified

7.1 In-Core HW-DIFT

Figure 4 shows a visual depiction of the in-core design with a five-staged pipeline of processor with the logic added to perform DIFT operations. To support the data flow tracking, the general-purpose register file and special function registers are extended to include a tag. As tags must also be propagated to the memory system, extensions to the data and instruction caches, system bus, and underlying memory storage are also required. The instruction fetch stage obtains code to execute from the instruction cache (iCache) and sends the code to the decode stage of the CPU. The instruction decoder then sends instruction-related information to a tag propagation module at the execute stage. This information can contain register indexes and the instruction type. In the execute stage, the memory instruction is processed to send requests to the memory system. The data tag is exchanged with the memory system according to the instruction type. Before the instruction can be committed in the write-back stage, it needs to be checked according to the specified tag check rule. Some of the in-core designs [23, 58] integrate the tag check logic within the write-back stage. Alternatively, the tag logic can also be implemented as an independent pipeline stage to reduce the stall cycles.

Because of the close integration of DIFT operations into the pipeline in the in-core design, security policies can be enforced as instructions execute. This allows the prevention of attacks that violate such policies because the information flow rules are checked before instructions are committed, with exceptions raised when instructions violating the policies are flagged. However, the in-core design requires significant modifications on the processor, which are expensive to integrate with commercial cores, thus requiring vendor support.

The earliest known example of a real-world DIFT processor is Elbrus E2K 512-bit wide VLIW microprocessor developed by the Moscow Center of SPARC Technologies (MCST) and fabricated by TSMC in 2000 [5]. It supports two instruction set architectures (ISA): Elbrus VLIW and Intel x86. Elbrus 2000 introduced a dynamic data type-checking during execution to prevent unauthorized access: each pointer has additional type information that is verified when the associated data is accessed.

In 2004, Chow et al. [17] proposed TaintBochs, an emulated online heavyweight whole-system HW-DIFT based on the Intel x86 32-bit emulator Bochs to analyze the lifetime of sensitive data tainting, where the propagation occurs

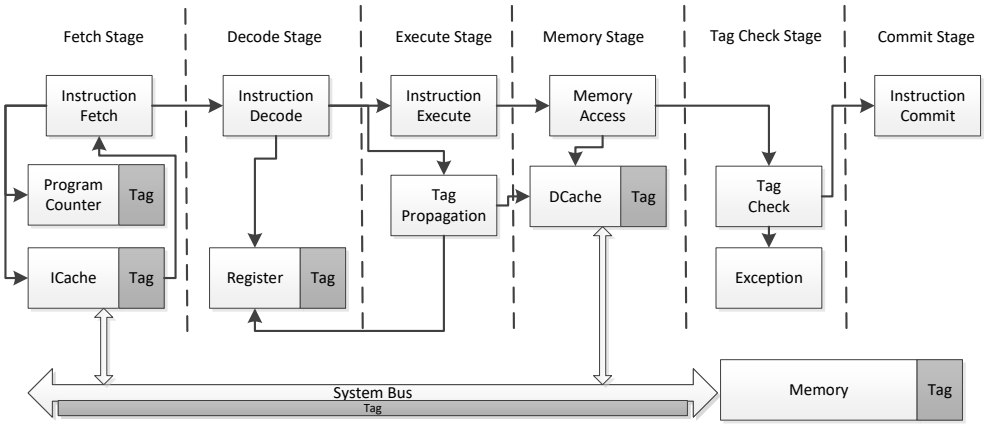

Fig. 4. In-Core Design Framework.

across the OS, programming language, and application boundaries. TaintBochs targets maintaining confidentiality by tracking sensitive data (e.g., passwords, credit card numbers) at byte-level granularity with tags stored in the guest's OS memory.

Concurrently, Crandall et al. [21] proposed Minos, a whole-system DIFT architecture for detection and mitigation of control flow attacks. Also implemented in Bochs, Minos utilizes word-granularity tags in memory by extending each word in memory by 1-bit. In contrast to TaintBochs, Minos targets maintaining Biba's low water-mark integrity policy across all individual words of data with low performance overhead ($< 10\%$).

Unlike most prior works, Tiwari et al. [75] introduced a user-agnostic architectural design of a processor with the goal to track all information flows within the machine, including all direct and indirect flows. The proposed design is based on the notion of Gate Level Information Flow Tracking (GLIFT) that can precisely augment logic blocks with tracking logic to support DIFT operations at the gate level. The implementation of their hardware design resulted in a custom ISA based on the GLIFT logic, which facilitates programmable policy enforcement.

Palmiero et al. [58] implemented a DIFT framework on a RISC-V processor core and synthesized it on a Field Programmable Gate Array (FPGA) board with a focus on IoT applications. To support different data-flow operations, the authors modified the processors' pipeline by introducing an additional pipeline stage before the instruction commit stage. The added new stage performs tag checking, allowing the design to detect any potential violation of security policies. The proposed design utilizes a byte-level tag, in which every byte of the data in memory and general purpose register is assigned a 1-bit tag. To support flexible memory protection, new custom instructions are added to set the tag of a specified memory data, e.g., setting the tag of a specified register or memory location. These custom instructions can be used to assign data with different tag values to enforce different security properties.

Tiwari et al. [74] enforced strong isolation between trusted and untrusted programs through information flow tracking. The authors introduced a new call-and-return mechanism (Execution Leases), where the caller function enforces a bound on the address space accessible to the callee function. This work uses tags to represent whether the corresponding action should be "admitted" or "denied". Additionally, new ISAs are introduced to support setting memory access bounds.

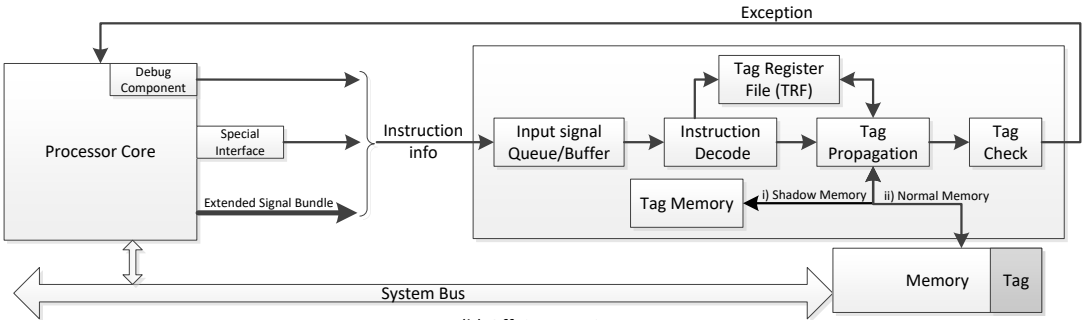

Fig. 5. Off-Core Design Framework.

7.2 Off-Core HW-DIFT

Figure 5 presents a basic *off-core* design. Unlike the in-core design, the instruction related information is sent to the processor's external sink (e.g., a DIFT-supported coprocessor) from the main processor for analysis. The DIFT coprocessor generally includes its own instruction decoders to aid with data flow tracking. Tag storage can be implemented with either a shadow memory or the system's memory. The latter requires support to control access to tag information by applications. In certain implementations, the processor is slightly modified to output internal states to the coprocessor.

The main difference between the in-core and the off-core designs is that the latter introduces a latency between attack incursion and detection. In the off-core design, all committed instructions are sent to the coprocessor to be processed later. At this point in time, these instructions have already been executed and the detrimental effects of a potential attack may have already been materialized, which is of great concern for attacks that attempt to compromise confidentiality, such as sensitive data leaks.

Similar to Raksha's design (see Section 6.2), Kannan et al. [41] introduce a coprocessor with a coarse-grained synchronization scheme, which allows for the processor to temporarily fall behind the main core in execution and synchronize only on system calls with 0.79% performance overhead. The custom interface between the processor and the DIFT coprocessor passes instruction encoding along with committed PCs and memory addresses. Lee et al. [46, 47] proposed the use of a debug interface to extract essential information, such as process ID, PC, and memory addresses from the main processor and output a signal bundle to support DIFT operations with 1.6% slowdown. A dedicated coprocessor receives execution information from the signal bundle to perform the desired DIFT operations. Similarly, Wahab et al. [79, 80] used the existing infrastructure of the ARM CoreSight architecture to develop a DIFT framework. The ARM CoreSight provides a debug interface allowing the live extraction of program execution and, in some instances, traces. The trace information is sent to a CoreSight sink for further processing. The information obtained from the sink is then sent to a dedicated DIFT coprocessor, which analyzes the instruction trace and propagates tags.

The processor and the sink communicate via either a debug component (e.g., ARM Core- Sight TPIU) or a special coprocessor interface (e.g., Rocket Custom Coprocessor -RoCC-in the Rocket-Chip). The greatest difference between these two types of channels is whether they can recover the whole execution states of an instruction in the processor. The ARM CoreSight component, for example, cannot acquire the memory address of memory instructions without binary instrumentation.

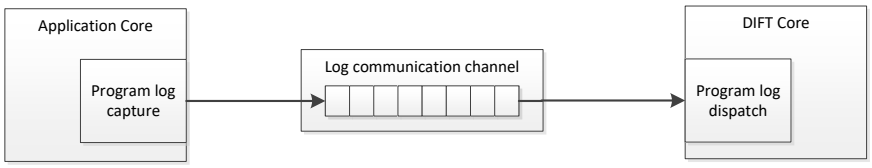

Fig. 6. Off-Load Design Framework.

7.3 Off-Load HW-DIFT

In an off-load design [14, 55] (depicted in Figure 6), the target program is executed in one processor core while the corresponding DIFT operations are instrumented in another. Unlike the off-core design, the off-load design needs a dedicated processor core for DIFT operations and can only be implemented in a multi-core processor, where the two cores need to communicate (e.g., shared memory) so that taint tracking operations are implemented.

This design does not require substantial modifications in the target hardware platform, but high latency is incurred due to the need of synchronization between the target and the monitoring program (representing DIFT operations) running on different cores and the need to instrument the target program for DIFT operations.

For the off-load design, the likelihood of attack prevention depends on the design and the performance overhead requirements. For example, an attack can be prevented if the target program waits for the check results from the DIFT core. The corresponding operation is stopped once the check fails. On certain implementations of this design, for performance considerations, the target program does not wait for check results and continues to execute until an interrupt is raised (check fails). In this case, the effects of the attack might persist even after its detection.

Nagarajan et al. [55] proposed an off-load based information flow tracking design, where the related execution information is exchanged between the target thread on the main core and the DIFT thread on the additional core. The authors compared two communication channels: software-based channel (via shared memory) and hardware-based channel (through a hardware based FIFO buffer). The software-based channel causes high performance overhead on both inter-thread (caused by cache misses) and intra-thread (to synchronize the pace of two programs). For the hardware-based channels, there is no need to synchronize the target and the DIFT threads. The incurred software performance overhead can be ameliorated by optimizing the target program and the modified program where taint tracking operations occur. Design schemes without hardware queue usually cause a slowdown $< 4x$, while those with hardware queue usually cause a slowdown of $0.5x$.

7.4 Summary

There are three design choices for HW-DIFT– in-core, off-core, and off-load (see comparison of performance and hardware overhead for different types of hardware based DIFT designs in Table 2). Amongst the in-core HW-DIFT implementations, the reporting of performance and hardware overhead metrics vary, with half of the implementations not reporting specifics on performance overhead [17, 58, 75]. This is likely due to the increased complexity of the hardware designs causing typical metrics to be less useful. For those works that do report metrics [21, 23, 74], we see a wide range of performance overheads [21, 23, 74], a slight increase in storage overhead [23, 58, 74], and multiple solutions with significant increases to hardware overhead in the form of lookup tables [58, 74, 75]. The in-core solutions offer streamlined DIFT solutions at the cost of significantly increased hardware complexity [17, 21, 23, 58, 74, 75]. In comparison, both the off-core [41, 46, 47, 79, 80] and off-load [14, 55] DIFT solutions provide significantly less

hardware complexity. Although the off-core DIFT solutions report increased hardware overhead, this is a result of these solutions introducing an entirely separate coprocessor. The off-core DIFT solutions introduced techniques that utilize debug interfaces to extract crucial information for a separate DIFT coprocessor to operate on. For example, Lee et al. [46, 47] first proposed the usage of a debug interfaces to communicate essential information to a coprocessor for DIFT operations, with Wahab et al. [79, 80] using the existing debug interface provided by the ARM CoreSight architecture. Contrasting, off-load DIFT solutions provide the simplest hardware implementation, as they typically do not require any hardware modifications at all, as seen in Chen et al. [14] and Nagarajan et al. [55]. However, we notice that these off-load DIFT designs report the highest performance latencies of all of the hardware based information flow tracking techniques covered in our survey. Both of these off-load DIFT implementations [14, 55] do not specify any hardware overhead metrics, as the changes necessary to the existing hardware platforms are negligible.

Several factors influence the performance overhead and the applicability of these designs, such as the target ISA, the complexity of hardware modifications, tag storage, and the information needed to support the DIFT operations. Generally, no software instrumentation is needed for in-core designs to operate, which results in no noticeable software performance overhead. This type of design also has the advantage of preventing attacks because instructions are only committed after the DIFT checks. However, this design requires significant and complex hardware modification, which impacts deployability due to the need of vendor support for DIFT boutique processor lines The off-core design requires relatively simple hardware modification because DIFT operations are delegated to a coprocessor, which can be easily repackaged and deployed alongside existing processor IP, thus facilitating deployability. However, because DIFT operations occur in parallel to CPU operations, it may not be possible to detect an ongoing attack until a portion of it has taken effect. The off-core design may also need some software communication with the DIFT core.

Lastly, the off-load design, which is suitable for multi-core systems, requires no hardware modifications, but involves software instrumentation, incurring non-negligible performance overheads due to the communication the cores. Exchange of information between the core running the program and the core performing taint tracking operations is often processed using auxiliary software. If real-time tracking of information flow is desired, the software being examined may need to be modified to allow for synchronous DIFT operation. This incurs high software performance overhead, which is further exacerbated by latencies associated with the software collecting and processing information flow metadata.

In addition to tracking information flow on general-purpose computing architecture, off-core designs can be customized to support emerging computing architectures including GPUs and other hardware accelerators. Different from general computing architectures, these parallel computing architectures adopt small caches, simple control unit and energy efficient ALU to boost throughput. The off-core design on general computing architectures uses the essential information extracted from the CPU to recover the instruction trace and analyze possible security violations. Therefore, DIFT components will need to be customized when applied to new computing architectures. For example, GPUs adopt massive threads to tolerate latency. In this case, the granularity of tags should be determined according to the numbers of threads instead of the width of the data bus and memory units. Also, the main security rules on parallel computing architectures should also be updated, e.g., enforcing isolation on data from different threads or different application scenarios.

8 DISCUSSION, LIMITATIONS, AND FUTURE DIRECTIONS

Table 1 provides a compilation of all works systematized in this paper. Table 2 focuses on providing a summary on works implemented entirely in hardware and categorized by different design choices. Table 3 summarizes the different

design types of DIFT proposals based on the platform for implementation – SW-DIFT, HW-DIFT, or SW-HW DIFT, where DIFT is implemented entirely in software, entirely in hardware, or collaboratively in software and hardware, respectively.

Table 1 categorizes as DIFT Fundamentals those works that best illustrate the impact of different attributes on the performance and accuracy of DIFT systems. Some implementations showcase different attributes in prototypes where DIFT is applied in the whole-system [50, 65, 67, 69, 71] or in a particular application [1]. Only a few works attempt to track indirect flows and/or address the overtainting vs undertainting dilemma [30, 65, 69].

SW-DIFT works can be targeted for *online* (e.g., [19, 71]) or *offline* operation (e.g., [4, 30, 65]) - see Tables 1 and 3. Both approaches offer flexibility in implementation of tag sizes, metadata granularity, and security policies. Nevertheless, SW-DIFT suffers from substantial performance overheads due to their reliance on binary translations and program instrumentation, which makes DIFT on-the-fly usage for security policies enforcement not practical. Most SW-DIFT solutions use byte granularity (see Table 1) [4, 6, 12, 18, 28, 30, 37, 38, 40, 42, 51, 54, 56, 62, 83, 85] and focus on offline applications, such as reverse engineering [4, 30, 65] (also see Table 3).

SW-HW DIFT has the potential to reduce the substantial performance overhead incurred in entirely software implementations, while retaining some design flexibility. This can be achieved either through (see Table 3) *HW-Supported SW*, where an abstract ISA is built to handle DIFT operations and the hardware platform is customized to support the ISA, or *SW-Supported HW*, where hardware provides an architectural structure for fundamental taint tracking operations (e.g., tag storage, propagation, and checking), while the software subsystem supplements the hardware by handling more complex DIFT-related operations (e.g., security exceptions and tag storage allocation). Although SW-HW DIFT provides relatively better performance compared to SW-DIFT, it requires non-negligible modifications in both software and hardware subsystems, which adds challenges for real-world deployment.

HW-DIFT provides further reduction in performance overheads compared to its software counterparts, with currently three types of explored designs: *in-core*, where the main processor is modified to support DIFT operations, *off-core*, where the DIFT operations are transferred to a coprocessor, and *off-load*, where a separate processor core is designated for DIFT operations (see Tables 2 and 3). In-core design does not require software instrumentation and, as instructions are committed only after DIFT checking, it can prevent security policy violations. However, the in-core design requires complex hardware modifications relying on vendor support. The off-core design has better deployability potential but may occasionally need software instrumentation. One limitation is that it cannot prevent some effects of attacks as DIFT checking is performed after instructions have been committed. The off-load design, on the other hand, does not require any hardware modifications, but requires the target program to be instrumented and relies on synchronization between the main processor core and the dedicated DIFT processor, resulting in high performance overheads.

In contrast to most SW-DIFT solutions, for both SW-HW DIFT and HW-DIFT solutions that we included in this work, tag granularity varies, with many designs having per word [21–23, 47, 61] or configurable tag granularities [60, 79, 80] (see Table 1). We included many HW-DIFT solutions that are designed to significantly accelerate the processing of taint tracking operations in comparison to SW-DIFT solutions [10, 13, 17, 21, 22, 39, 41, 47, 49, 58–60, 74, 75, 79, 80].

The majority of HW-DIFT implementations utilize a 1-bit tag format [17, 21, 22, 47, 49], as it is the most straightforward way to extend hardware designs to handle DIFT tags. More recent HW-DIFT implementations have shown progress in limiting runtime and performance overheads in comparison to the majority of SW-DIFT implementations [13, 21, 22, 39, 41, 47, 49, 58] (Table 1).

Table 3. Summary of Systematization

	Category	Description	Advantages	Limitations
SW-DIFT	*Online*	- run-time tracking mechanism	- flexibility in implementation of DIFT fundamental units	- prohibitive performance overhead for real-time applications
	Offline	- record-and-play techniques - usually used for reverse engineering (e.g., malware/intrusion analysis)	- flexibility in implementation of DIFT fundamental units - can examine the program in detail without performance constraints	- substantial performance overhead - not suitable to enforce security policies on-the-fly
SW-HW DIFT	*HW-Supported SW*	- abstract ISA to handle DIFT operations - customized hardware platform to support the abstract ISA.	- better performance compared to purely SW-DIFT	- requires modifications in both SW and HW
	SW-Supported HW	- HW provides an architectural structure for fundamental DIFT operations - SW supplements by handling more complex DIFT operations		
HW-DIFT	*In-Core*	- Main processor is modified to support DIFT operations - Instruction is only committed after DIFT checking is performed	- no SW instrumentation needed - all security violations can be prevented	- requires complex HW modifications - needs vendor support
	Off-Core	- DIFT operations occur in a co-processor	- may need SW instrumentation - need relatively simpler HW modification - easy repackaging and better deployability	- some effects of attacks cannot be prevented as DIFT checking is performed after instruction commits
	Off-Load	- Involves a dedicated processor core for DIFT operations.	- does not require HW modifications	- target program needs to be instrumented - synchronization checks introduce latency resulting in high-performance overhead.

8.1 Limitations

For our study, we chose to focus on dynamic information flow tracking specifically, rather than information flow in general. For example, there has been a large body of work on dynamic binary instrumentation [24] and other static approaches to information flow tracking at design or compilation time [48, 52]. DIFT encompasses a distinct problem space from other information flow tracking approaches because it seeks to be fully dynamic. This means that information flow reasoning is based on program traces, not higher-level program constructs that give information about code that does not run. It also means that DIFT can be applied to commodity systems, even when the source code is not available. Closed source software, malware, and cloud systems, where users are free to install and configure any software they wish, are examples where a system may want to apply information flow tracking without having access to all the source code. Furthermore, even when all source code is available, it alone does not describe a full, live system. Modern systems operate via a large amount of dynamic loading, inter-process communication, memory sharing, and other low-level abstractions that are naturally captured by DIFT, but make static analysis before compiling the system prohibitively complex. Because we are interested in specifically the problem space that DIFT performs well in, and also because we want our results and analysis to be applicable to this problem space, we limited the set of papers that we considered mostly to DIFT papers. Furthermore, we limited the systematization to only DIFT schemes, and did not include flow controlling designs (e.g., HiStar [84]), or other language-based approaches.

Lastly, although we focused on security models handling confidentiality and integrity, such as those of, respectively, Bell-LaPadula [7, 45] and Biba [8], we acknowledge that there are other models used for enforcing information flow security, which were not covered by the works systematized in this paper. One example is the non-interference model by Goguen and Meseguer [33], which introduces the notion that the manipulation of private information has no effect on public observations of data and is commonly used for specification of policies in language-based security [63].

8.2 Recommendations

Integrating Static Approaches to DIFT. DIFT only detects unallowable information flows if they occur while the program is running. Static information flow mechanisms, on the other hand, can anticipate unauthorized flows prior

to execution. We believe that integrating static and dynamic approaches could significantly improve the security of information flows in the system, potentially helping to alleviate the need to track all indirect flows at run-time. Few proposals in the literature explore hybrid static and dynamic approaches. For example, Kang et al. [40] use static analysis at compilation time for information flow analysis, with performance overheads application-dependent. Another example is the work from Wahab et al. [79, 80], a software-assisted hardware DIFT that performs static analysis during compilation time for tag annotations. While SW-HW DIFT systems may be expensive to implement at the whole system level , specialized applications could be useful for integrating static and dynamic techniques to address indirect information flows. Based on our findings, most proposals in the literature do not explore hybrid approaches (static and dynamic). Researchers either focused on hardware optimizations, software techniques for application specific designs, or considered static analysis too expensive or incomplete, thus favoring an entirely dynamic system. The scarcity of literature in hybrid systems motivates further exploration.

Handling Indirect Flows vs Overhead. To create an effective DIFT system, indirect flows must be correctly tracked and propagated. Our systematization did not find any system that can fully track indirect flows without noticeable performance issues. Tracking indirect flows is non-trivial, which has lead to diverse solutions at various layers of abstraction. Tiwari et al. [75] attempt to handle indirect flows using a constrained ISA. The V-DIFT system from Espinoza et al. [30] defines taint vectors for tags and works at the tag level of abstraction. However, current approaches are not able to track indirect flows completely without incurring over-tainting. Wahab et al. [79] used a custom ISA and binary translation to handle indirect flows while maintaining modest overheads ranging from 5% to 24%. However, their system's translations might not be appropriate in all cases. The Neutaint system from She et al. [67] attempts to select optimal taint propagation policies using a neural network. Sapountzis et al. [65] provide valuable insight into handling indirect flows analytically and, although the problem has been observed to be NP-hard, their work provides evidence that a relaxed version of the problem can aid in handling overtainting. We suggest continuing to explore analytical optimization solutions for indirect flow propagation, particularly in hardware. Table 1 provides details on the systems we found that track indirect flows, regardless of their handling of over-tainting and their incurred runtime overhead.

A key question is: is it possible to handle indirect flows and maintain acceptable performance overhead simultaneously? Our systematization shows that research efforts have focused on addressing either runtime performance or indirect flows in isolation. However, DIFT solutions cannot be practical (performance) or effective/accurate (handling indirect flows) without addressing both issues. Our systematization indicates that handling indirect flows requires propagating more tags and potentially different tag types. Most work have attempted to handle taint and tracking operations in ways that are sequential and synchronized with the target program or system. One avenue not pursued for DIFT is **parallelism**, especially in hardware. We envision that research focused on this vein could yield results useful for addressing runtime performance and indirect flow tracking requirements simultaneously. More specifically, DIFT systems could be built in ways that capture indirect flows (e.g., more tags, more complex tags), while incurring acceptable performance overheads utilizing hardware parallelism. Another research direction not fully explored is temporal and spatial locality for tag creation and propagation, which could alleviate performance costs and the need to modify existing hardware subsystems to accommodate DIFT.

Standardized benchmarks. A major need for the research community to make progress in bringing DIFT into practice is a common set of benchmarks, that can enable direct comparisons between different DIFT systems. This will be very challenging, as our systematization of the DIFT literature has shown that there are many different ways that DIFT is implemented and applied. Moreover, our survey has also revealed another research gap: there are published efforts that

evaluate the performance overhead of DIFT and show acceptable performance to be used in practice, and there are published efforts that show the efficacy of DIFT for real-world, challenging programs that include indirect flows, but not much at the intersection of these two prerequisites for a real-world, practical DIFT system. A set of benchmarks addressing both performance and the ability to track indirect flows of information would help to fill this gap, and would give DIFT research efforts a common ground to compare different approaches more directly.

Promoting Hardware Implementation. Our systematization shows that SW-DIFT is appealing as a reverse engineering tool, but impractical for on-the-fly operation due to the prohibitive performance overhead. Table 1 provides runtime overheads for the various SW-DIFT, SW-HW DIFT, and HW-DIFT. Notice that in general, SW-DIFT incur significant runtime overhead (several orders of magnitude above a given applications base run-time) when compared to HW-DIFT platforms. Alleviating, if not eliminating, the performance overhead without dedicated hardware is difficult. However, HW-DIFT options are not yet fully mature in that proposed prototypes still require substantial changes in the processor and hardware and software subsystems. Our systematization indicates promise in the off-core design, where the coprocessor is entirely dedicated to DIFT operations, to further enhance the performance of taint tracking operations and make fabrication and real-world deployment feasible. Off-core designs (e.g., decoupled DIFT coprocessor [41, 46, 47, 79, 80]) have shown considerable potential for DIFT as a built-in security system for devices with relatively simple security policies (e.g., prevention of stack-based buffer overflows). Given this, we advocate a focus on off-core designs.

Processing DIFT operations in an off-core manner, however, raises concerns around sharing information between the core processor and the entity (e.g., coprocessor) performing the DIFT operations. Nevertheless, we forsee it to be a promising architectural design choice for DIFT in hardware, thus promoting the widespread use of DIFT. The breadth of options to test and evaluate the effectiveness of DIFT further highlights the body of research yet to be performed. As the field progresses, we expect hardware implementations to continue to target real-time program analysis rather than offline record-and-replay, as this would allow for detection and mitigation as threats occur.

DIFT in the IoT Landscape. On the surface, DIFT seems to be at odds with the IoT landscape, due to DIFT implementations still incurring performance overhead and the resource constraints of IoT devices. However, we observe that DIFT, if implemented with minimal hardware overhead, could be a game-changer for IoT ecosystems. Specifically, as the off-core design for DIFT in hardware continues to advance, manufacturers could begin to ship their products with a configurable DIFT architecture directly built into their devices because this particular design has the potential to package DIFT into a simple and easy-to-integrate hardware module. This transition seems like a natural fit given that the current generation of IoT devices typically lack dedicated security mechanisms whether that be in their hardware or the software aboard. We predict that DIFT will fill the role as a built-in, dedicated security mechanism that facilitates this “plug and play” feature; a highly desirable feature in the IoT paradigm.

Confidentiality and Integrity Policies There are a wide variety of DIFT implementations focused on enforcing either confidentiality or integrity policies. However, we did not find any implementation that attempts to enforce both or study the effects of both policies operating in tandem. Given the prevalence and severity of attacks targeting these two CIA pillars, we recommend additional research on architectures enforcing both policies without conflicts, given that integrity policies in many cases are the dual of confidentiality policies.

9 CONCLUSION

In this paper, we systematized literature on dynamic information flow tracking (DIFT), compiling common knowledge and formulating research gaps between the existing literature and real-world requirements for DIFT implementation. To inform on DIFT foundations prior to our systematization, we briefly discussed fundamentals of DIFT, including tag attributes (size, format, and granularity), tag insertion, tag propagation, and tag checking. We provided an exposition of DIFT from two core dimensions – i) the layer of abstraction for the DIFT implementation: software, hardware, and their combination, and ii) the security goal: confidentiality and/or integrity. We identified two major challenges that impede the practical application of DIFT for real-world deployment – i) prohibitive performance and memory overhead, i.e., high memory and runtime costs, and ii) the non-optimal tracking of information that leads to undertainting vs overtainting dilemma (low accuracy). We believe that addressing these two major limitations for DIFT via hardware implementation using the off-core design and exploring parallelism and temporal and spatial locality will unleash DIFT's great potential for on-the-fly security policy enforcement, especially for the IoT ecosystem, as DIFT can streamline the implementation of security policies in a built-in and standardized fashion. Based on the insights from our systematization and to make DIFT practical for real-time applications, we provided various recommendations for the next generation of efficient DIFT systems.

10 ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grants No. 1801599, 1801613, 1801593, and 2007741. This material is based upon work supported by (while serving at) the National Science Foundation.

REFERENCES

[1] Mohammed I. Al-Saleh and Jedidiah R. Crandal. 2010. Tracking Address and Control Dependencies for Full-System Information Flow Measurement. Technical Report. University of New Mexico, Department of Computer Science.

[2] Alexandre Becoulet. 2021. The MutekH Project. http://www.mutekh.org/

[3] Saleh Mohamed Alnaeli, Melissa Sarnowski, Md Sayedul Aman, Ahmed Abdelgawad, and Kumar Yelamarthi. 2017. Source Code Vulnerabilities in IoT Software Systems. Advances in Science, Technology and Engineering Systems Journal 2, 3 (2017), 1502–1507.

[4] Meisam Navaki Arefi, Geoffrey Alexander, Hooman Rokham, Aokun Chen, Michalis Michalis Faloutsos, Xuetao Wei, Daniela Oliveira, and Jedidiah R. Crandall. 2018. Faros: Illuminating In-Memory Injection Attacks via Provenance-based Whole-system Dynamic Information Flow Tracking. In Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). 231–242. https://doi.org/10.1109/DSN.2018.00034

[5] Boris Babayan. 2000. E2K Technology and Implementation. In Proceedings of the European Conference on Parallel Processing. Springer, 18–21. https://doi.org/10.1007/3-540-44520-X_2

[6] Subarno Banerjee, David Devecsery, Peter M. Chen, and Satish Narayanasamy. 2019. Iodine: Fast Dynamic Taint Tracking Using Rollback-free Optimistic Hybrid Analysis. In Proceedings of the IEEE Symposium on Security and Privacy. 490–504. https://doi.org/10.1109/SP.2019.00043

[7] D. Elliott Bell and Leonard J. LaPadula. 1976. Secure Computer System: Unified Exposition and Multics Interpretation. Technical Report. The MITRE Corporation.

[8] Ken Biba. 1975. Integrity Considerations for Secure Computer Systems. The MITRE Corporation (1975).

[9] Matthew Bishop. 2019. Computer Security: Art and Science. Addison-Wesley Pearson, Boston.

[10] Erik Bosman, Asia Slowinska, and Herbert Bos. 2011. Minemu: The World's Fastest Taint Tracker. In Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID). Springer-Verlag, 1–20. https://doi.org/10.1007/978-3-642-23644-0_1

[11] David Budgen and Pearl Brereton. 2006. Performing Systematic Literature Reviews in Software Engineering. In Proceedings of the 28th ACM International Conference on Software Engineering (ICSE). 1051–1052. https://doi.org/10.1145/1134285.1134500

[12] Walter Chang, Brandon Streiff, and Calvin Lin. 2008. Efficient and Extensible Security Enforcement Using Dynamic Data Flow Analysis. In Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS). 3950. https://doi.org/10.1145/1455770.1455778

[13] Kejun Chen, Qingxu Deng, Yumin Hou, Yier Jin, and Xiaolong Guo. 2019. Hardware and Software Co-Verification from Security Perspective. In 20th IEEE International Workshop on Microprocessor/SoC Test, Security and Verification (MTV). 50–55. https://doi.org/10.1109/MTV48867.2019.00018

[14] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. 2008. Flexible Hardware Acceleration for Instruction-Grain Program Monitoring. In Proceedings of the ACM International Symposium on Computer Architecture (ISCA). 377–388. https://doi.org/10.1109/ISCA.2008.20

[15] Yu-Yuan Chen, Pramod A. Jamkhedkar, and Ruby B. Lee. 2012. A Software-Hardware Architecture for Self-Protecting Data. In Proceedings of the ACM Conference on Computer and Communications Security (CCS). 1427. https://doi.org/10.1145/2382196.2382201

[16] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. 2006. TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC). 749–754. https://doi.org/10.1109/ISCC.2006.158

[17] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. 2004. Understanding Data Lifetime via Whole System Simulation. In Proceedings of the 13th USENIX Security Symposium. 22.

[18] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: A Generic Dynamic Taint Analysis Framework. In Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA). 196–206. https://doi.org/10.1145/1273463.1273490

[19] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. 2005. Vigilante: End-to-End Containment of Internet Worms. In Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP). 133–147. https://doi.org/10.1145/1095810.1095824

[20] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. 2003. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]. 227–237. https://doi.org/10.1109/FITS.2003.1264935

[21] Jedidiah R. Crandall and Frederic T. Chong. 2004. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In 37th ACM International Symposium on Microarchitecture (MICRO). 221–232. https://doi.org/10.1145/1187976.1187977

[22] Jedidiah R. Crandall, S. Felix Wu, and Frederic T. Chong. 2006. Minos: Architectural Support for Protecting Control Data. ACM Transactions on Architecture and Code Optimization (TACO) 3, 4 (2006), 359–389. https://doi.org/10.1145/1187976.1187977

[23] Michael Dalton, Hari Kannan, and Christos Kozyrakis. 2007. Raksha: A Flexible Information Flow Architecture for Software Security. In Proceedings of the 34th Annual ACM International Symposium on Computer Architecture (ISCA). 482–493. https://doi.org/10.1145/1250662.1250722

[24] Daniele Cono D'Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro, and Lorenzo Cavallaro. 2019. SoK: Using Dynamic Binary Instrumentation for Security (And How You May Get Caught Red Handed). In Proceedings of the ACM Asia Conference on Computer and Communications Security (Asia CCS). https://doi.org/10.1145/3321705.3329819

[25] Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. Communications of the ACM 19, 5 (1976), 236–243.

[26] Dorothy E. Denning and Peter J. Denning. 1977. Certification of Programs for Secure Information Flow. Communications of the ACM 20, 7 (1977), 504–513.

[27] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. 2015. Repeatable Reverse Engineering with PANDA. In Proceedings of the 5th ACM Program Protection and Reverse Engineering Workshop (PPREW). 1–11. https://doi.org/10.1145/2843859.2843867

[28] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. 2007. Dynamic Spyware Analysis. In Proceedings of the USENIX Annual Technical Conference.

[29] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. ACM Transactions on Computer Systems (TOCS) 32, 2 (2014), 1–29.

[30] Antonio M Espinoza, Jeffrey Knockel, Pedro Comesaña-Alfaro, and Jedidiah R Crandall. 2016. V-Dift: Vector-based Dynamic Information Flow Tracking with Application to Locating Cryptographic Keys for Reverse Engineering. In 11th IEEE International Conference on Availability, Reliability and Security (ARES). 266–271. https://doi.org/10.1109/ARES.2016.97

[31] Jeffrey Stewart Fenton. 1973. Information Protection Systems. Ph.D. Dissertation. University of Cambridge.

[32] Jeffrey Stewart Fenton. 1974. Memoryless Subsystems. The Computer Journal 17, 2 (1974), 143–147.

[33] J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. In Proceedings of the IEEE Symposium on Security and Privacy. 11–11. https://doi.org/10.1109/SP.1982.10014

[34] Michael Gregg. 2013. CISSP Exam Cram. Pearson IT Certification, Indianapolis, Ind.

[35] Nevin Heintze and Jon G Riecke. 1998. The SLam calculus: Programming with Secrecy and Integrity. In Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). 365–377. https://doi.org/10.1145/268946.268976

[36] Ralf Huuck. 2015. IoT: The Internet of Threats and Static Program Analysis Defense. In EmbeddedWorld 2015: Exibition & Conferences. 493.

[37] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. 2017. Rain: Refinable Attack Investigation with on-demand Inter-process Information Flow Tracking. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS). 377–390. https://doi.org/10.1145/3133956.3134045

[38] Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee. 2018. Enabling Refinable Cross-host Attack Investigation with Efficient Data Flow Tagging and Tracking. In Proceedings of the 27th USENIX Security Symposium. 1705–1722.

[39] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W. Moore, Alex Bradbury, Hongyan Xia, Robert N.M. Watson, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alfredo Mazzinghi, Alex Richardson, Stacey Son, and A. Theodore Markettos. 2017. Efficient Tagged Memory. In Proceedings of the IEEE International Conference on Computer Design (ICCD). 641–648. https://doi.org/10.1109/ICCD.2017.112

[40] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In Proceedings of the Network and Distributed System Security Symposium, (NDSS). The Internet Society.

[41] H. Kannan, M. Dalton, and C. Kozyrakis. 2009. Decoupling Dynamic Information Flow Tracking with a Dedicated Coprocessor. In IEEE/IFIP International Conference on Dependable Systems Networks (DSN). 105–114. https://doi.org/10.1109/DSN.2009.5270347

[42] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. 2012. Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. ACM SIGPLAN Notices 47, 7 (2012). https://doi.org/10.1145/2365864.2151042

[43] Barbara Kitchenham, O. Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. 2009. Systematic Literature Reviews in Software Engineering – A Systematic Literature Review. Information and Software Technology 51, 1 (2009), 7–15. https://doi.org/10.1016/j.infsof.2008.09.009 Special Section - Most Cited Articles in 2002 and Regular Research Papers.

[44] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, Frans Kaashoek, M, Eddie Kohler, and Robert Morris. 2007. Information Flow Control for Standard OS Abstractions. ACM SIGOPS Operating Systems Review 41, 6 (2007), 321–334.

[45] Leonard J. LaPadula and D. Elliot Bell. 1973. Secure Computer systems: A Mathematical Model. Technical Report. The MITRE Corporation.

[46] Jinyong Lee, Ingoo Heo, Yongje Lee, and Yunheung Paek. 2015. Efficient Dynamic Information Flow Tracking on a Processor with Core Debug Interface. In Proceedings of the 52nd ACM Annual Design Automation Conference (DAC). https://doi.org/10.1145/2744769.2744830

[47] Jinyong Lee, Ingoo Heo, Yongje Lee, and Yunheung Paek. 2016. Efficient Security Monitoring with the Core Debug Interface in an Embedded Processor. ACM Transactions on Design Automation of Electronic Systems (TODAES) 22, 1 (2016), 1–29. https://doi.org/10.1145/2907611

[48] Yin Liu and Ana Milanova. 2010. Static Information Flow Analysis with Handling of Implicit Flows and a Study on Effects of Implicit Flows vs Explicit Flows. In 14th European Conference on Software Maintenance and Reengineering. 146–155. https://doi.org/10.1109/CSMR.2010.26

[49] Juan Carlos Martínez Santos and Yunsi Fei. 2013. Micro-Architectural Support for Metadata Coherence in Multi-Core Dynamic Information Flow Tracking. In Proceedings of the 2nd ACM International Workshop on Hardware and Architectural Support for Security and Privacy (HASP). https://doi.org/10.1145/2487726.2487732

[50] Juan Carlos Martinez Santos, Yunsi Fei, and Zhijie Jerry Shi. 2009. PIFT: Efficient Dynamic Information Flow Tracking Using Secure Page Allocation. In Proceedings of the 4th ACM Workshop on Embedded Systems Security (WESS). https://doi.org/10.1145/1631716.1631722

[51] Bita Mazloom, Shashidhar Mysore, Mohit Tiwari, Banit Agrawal, and Tim Sherwood. 2012. Dataflow Tomography: Information Flow Tracking For Understanding and Visualizing Full Systems. ACM Transactions on Architecture and Code Optimization 9, 1, Article 3 (March 2012), 26 pages. https://doi.org/10.1145/2133382.2133385

[52] Andrew C. Myers. 1999. JFlow: Practical Mostly-static Information Flow Control. In Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). 228–241. https://doi.org/10.1145/292540.292561

[53] Andrew C. Myers and Barbara Liskov. 2000. Protecting Privacy Using the Decentralized Label Model. ACM Transactions on Software Engineering and Methodology (TOSEM) 9, 4 (2000), 410–442.

[54] Shashidhar Mysore, Bita Mazloom, Banit Agrawal, and Timothy Sherwood. 2008. Understanding and Visualizing Full Systems with Data Flow Tomography. In Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). https://doi.org/10.1145/1346281.1346308

[55] Vijay Nagarajan, Ho-Seop Kim, Youfeng Wu, and Rajiv Gupta. 2008. Dynamic Information Flow Tracking on Multicores. Technical Report. University of Edinburgh.

[56] James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In The Network and Distributed System Security (NDSS) Symposium. The Internet Society.

[57] Meltem Ozsoy, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Tameesh Suri. 2011. SIFT: A Low-Overhead Dynamic Information Flow Tracking Architecture for SMT Processors. In Proceedings of the 8th ACM International Conference on Computing Frontiers (CF). 1–11. https://doi.org/10.1145/2016604.2016650

[58] Christian Palmiero, Giuseppe Di Guglielmo, Luciano Lavagno, and Luca P. Carloni. 2018. Design and Implementation of a Dynamic Information Flow Tracking Architecture to Secure a RISC-V Core for IoT Applications. In Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC). 1–7. https://doi.org/10.1109/HPEC.2018.8547578

[59] Luca Piccolboni, Giuseppe Di Guglielmo, and Luca P. Carloni. 2018. PAGURUS: Low-Overhead Dynamic Information Flow Tracking on Loosely Coupled Accelerators. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 37, 11 (2018), 2685–2696.

[60] Christian Pilato, Kaijie Wu, Siddharth Garg, Ramesh Karri, and Francesco Regazzoni. 2018. TaintHLS: High-level Synthesis for Dynamic Information Flow Tracking. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 38, 5 (2018), 798–808. https://doi.org/10.1109/TCAD.2018.2834421

[61] Joël Porquet and Simha Sethumadhavan. 2013. WHISK: An Uncore Architecture for Dynamic Information Flow Tracking in Heterogeneous Embedded SoCs. In Proceedings of the IEEE International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS). 1–9. https://doi.org/10.1109/CODES-ISSS.2013.6658991

[62] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. 2006. 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 (MICRO). 135–148. https://doi.org/10.1109/MICRO.2006.29

[63] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based Information-Flow Security. IEEE Journal on Selected Areas in Communications 21, 1 (2003), 5–19. https://doi.org/10.1109/JSAC.2002.806121

[64] Nikolaos Sapountzis, Ruimin Sun, and Daniela Oliveira. 2019. DDIFT: Decentralized Dynamic Information Flow Tracking for IoT Privacy and Security. In Proceedings of the Workshop on Decentralized IoT Systems and Security (DISS). The Internet Society. https://doi.org/10.14722/diss.2019.230007

[65] Nikolaos Sapountzis, Ruimin Sun, Xuetao Wei, Yier Jin, Jedidiah Crandall, and Daniela Oliveira. 2020. MITOS: Optimal Decisioning for the Indirect Flow Propagation Dilemma in Dynamic Information Flow Tracking Systems. In Proceedings of the 40th IEEE International Conference on Distributed Computing Systems (ICDCS). 1090–1100. https://doi.org/10.1109/ICDCS47774.2020.00093

[66] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (But Might Have Been Afraid to Ask). In Proceedings of the IEEE Symposium on Security and Privacy. 317–331. https://doi.org/10.1109/SP.2010.26

[67] Dongdong She, Yizheng Chen, Abhishek Shah, Baishakhi Ray, and Suman Jana. 2020. Neutaint: Efficient Dynamic Taint Analysis with Neural Networks. In Proceedings of the IEEE Symposium on Security and Privacy (SP). 1527–1543. https://doi.org/10.1109/SP40000.2020.00022

[68] SiFive. 2021. Freedom Platforms. https://github.com/sifive/freedom

[69] Asia Slowinska and Herbert Bos. 2009. Pointless Tainting? Evaluating the Practicality of Pointer Tainting. In Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys). 61–74. https://doi.org/10.1145/1519065.1519073

[70] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In Proceedings of the International Conference on Information Systems Security. Springer, 1–25. https://doi.org/10.1007/978-3-540-89862-7_1

[71] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. 2004. Secure Program Execution via Dynamic Information Flow Tracking. In Proceedings of the 11th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 85–96.

[72] Synopsys. 2019. Black Duck Audits. https://bit.ly/2N3bvWs

[73] Synopsys. 2020. Source Security and Risk Analysis (OSSRA) Report. https://www.techrepublic.com/resource-library/whitepapers/2020-open-source-security-and-risk-analysis-report/

[74] Mohit Tiwari, Xun Li, Hassan MG Wassel, Frederic T. Chong, and Timothy Sherwood. 2009. Execution leases: A Hardware-Supported Mechanism for Enforcing Strong Non-Interference. In Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). 493–504. https://doi.org/10.1145/1669112.1669174

[75] Mohit Tiwari, Hassan MG Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. 2009. Complete Information Flow Tracking from the Gates Up. In Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 109–120. https://doi.org/10.1145/2528521.1508258

[76] Daniel Townley, Khaled N. Khasawneh, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Lei Yu. 2019. LATCH: A Locality-Aware Taint CHecker. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). 969–982. https://doi.org/10.1145/3352460.3358327

[77] Manish Vachharajani, Neil Vachharajani, David A. Penry, Jason A. Blome, Sharad Malik, and David I. August. 2006. The Liberty Simulation Environment: A Deliberate Approach to High-Level System Modeling. ACM Transactions on Computer Systems 24, 3 (Aug. 2006), 211–249. https://doi.org/10.1145/1151690.1151691

[78] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. 2004. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). https://doi.org/10.1109/MICRO.2004.31

[79] Muhammad Abdul Wahab, Pascal Cotret, Mounir Nasr Allah, Guillaume Hiet, Arnab Kumar Biswas, Vianney Lapotre, and Guy Gogniat. 2018. A Small and Adaptive Coprocessor for Information Flow Tracking in ARM SoCs. In Proceeding of the IEEE International Conference on ReConFigurable Computing and FPGAs (ReConFig). 1–8. https://doi.org/10.1109/RECONFIG.2018.8641695

[80] Muhammad A Wahab, Pascal Cotret, Mounir N Allah, Guillaume Hiet, Vianney Lapotre, and Guy Gogniat. 2017. ARMHEx: A Hardware Extension for DIFT on ARM-based SoCs. In IEEE 27th International Conference on Field Programmable Logic and Applications (FPL). 1–7. https://doi.org/10.23919/FPL.2017.8056767

[81] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. 2019. TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V. In The Network and Distributed System Security Symposium (NDSS). The Internet Society.

[82] Wei Xu, Sandeep Bhatkar, and Ramachandran Sekar. 2006. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In Proceedings of the 15th USENIX Security Symposium. 121–136.

[83] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: Capturing System-Wide Information Flow for Malware Detection and Analysis. In Proceedings of the 14th ACM Conference on Computer and Communications Security. 116–127. https://doi.org/10.1145/1315245.1315261

[84] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2006. Making Information Flow Explicit in HiStar. In Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI).

[85] David Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. 2011. TaintEraser: Protecting Sensitive Data Leaks using Application-Level Taint Tracking. ACM SIGOPS Operating Systems Review 45, 1 (2011), 142–154. https://doi.org/10.1145/1945023.1945039