Principles of Data Flow Integrity: Specification and Enforcement

Toktam Ramezanifarkhani and
Mohammadreza Razzazi

Amirkabir University of Technology
Email: {t_ramezani, razzazi}@aut.ac.ir

Subverting runtime data flow is common in many current software attacks. Data Flow Integrity (DFI) is a policy whose satisfaction can prevent such attacks. This paper develops a formal foundation on DFI specification, and characteristics of its enforcement techniques with formulations of hypotheses and guarantees. Enforcement techniques are based on static analysis and program monitoring at runtime. This foundation can be used for practical satisfaction of DFI and help establish guarantees in every applied platform.

Keywords: Data Flow Analysis, Data Flow Integrity, Reference Monitor, Security Policy Enforcement

[ ]

1. INTRODUCTION

Data flow subversion at runtime is a common step of abundant security attacks [1, 2]. Despite previous research on techniques to prevent such attacks, they are still among the most critical security attacks and software is likely to remain vulnerable to them in the future [3, 4, 1, 2]. We believe that it is due to the lack of general and platform-independent specification and enforcement of DFI, unclear hypothesis and assumptions (as we can see in related works in Section 5), and vulnerability-based mitigation techniques. These all result in less precision in the enforcement techniques with possibility of circumvention, and make their evaluations and effectiveness measurements harder.

These attacks violate Data Flow Integrity (DFI) that briefly imposes restrictions on runtime data flows to be allowed by program data flow graph. DFI is firstly defined in [1] informally and its definition was more according to a specific implementation than a general definition. However, a more general implementation-independent and platform-independent specification of the policy with explicit assumptions such as an expressive formal study on DFI is missed in previous works.

In security, it is important to identify assumptions to specify policies and evaluate provided enforcement techniques. Therefore, it is necessary to make assumptions clear and well-defined because an attacker that can invalidate assumptions can often circumvent the enforcement. Since in security any set of assumptions is likely to be incomplete, clarifying them makes it simpler to extend or improve the specification and enforcement of a desired policy by completing the list of assumptions, or providing their satisfactions instead of just assuming them in further researches. Hence, dealing with DFI in this paper, we focus on clarifying assumptions and hypothesis that we consider most relevant.

Violations from DFI usually happen because of exploiting software vulnerabilities, such as buffer overflows or format strings, which enable attackers to overwrite arbitrary memory locations. Therefore, vulnerability mitigation techniques such as static or dynamic elimination of buffer overflows, e.g. [5], program protection

from format string attacks, e.g. [6], and many other techniques are useful to prevent attacks against DFI. Nevertheless, all vulnerabilities that can result in DFI violations are not mitigated yet [2, 7, 4]. Moreover, often such vulnerability-based techniques not only are not applicable to mitigate all exploitable, documented and undocumented vulnerabilities but also just target specific classes of a vulnerability, e.g. size sensitive techniques to overflows [8]. Furthermore, because of false positive and/or false negatives, techniques and tools provide imperfect approaches that can be defeated or circumvented [9]. In this paper, we do not focus on these vulnerabilities or a class of them. Instead, we are concerned with the foundations of DFI specification and enforcement that in addition to overcome such challenges can also be considered as a complementary approach in case of applied vulnerability-based techniques.

This paper develops vulnerability, implementation and platform-independent specification of DFI. Then, inductive definition of DFI is developed by inference rules that underlies techniques for DFI enforcement dynamically at runtime. To describe requirements of DFI enforcement and justify its applicability, we propose an enforcement approach based on Reference Monitoring (RM) [10, 11] and a combination of Software Memory Access Control (SMAC) [12] and Control Flow Sequentiality (CFS). RMs, as software security policy enforcement mechanisms, are widely used and practical in many modern and security critical systems because of the ever-increasing need to address defects and security attacks of software systems at runtime. Further, SMAC is an efficient variant of software fault isolation which is embodied in reference monitor for access to memory regions, and is helpful in RM implementation. Moreover, CFS is a security property that restricts the sequence of instruction in a program execution to the program control flow graph. CFS can prevent the circumvention of these two enforcement mechanisms, i.e. RMs and SMAC and also, as we explain in this paper, is useful in DFI enforcement. Characteristics of a DFI enforcement technique, based on these mechanisms and properties, in addition to a theorem about the program executions after applying such enforcement techniques are explained. However, due to ruling out policy violation at runtime, dynamic security policy enforcements as more precise, popular and widely used policy enforcement approaches come with its price of overhead.

We consider this basic, general and formal study of the DFI policy specification and enforcement as a major difference with previous works, and as an important similarity with research on type-safe languages. Furthermore, our formal study is useful not only because of clarifying hypotheses and guarantees, but also as a guide in design and development of customized enforcement and implementation for every platform with variety of features, e.g. underlying operation system, supported language features and system encountering risk level and tolerance. To the best of our knowledge, this is the first approach that formalizes DFI.

Section 2 describes preliminaries and specification of the DFI policy. In Section 3, inductive definition of DFI is developed by inference rules. Basics of DFI enforcement and related theory are discussed in Section 4. Section 5 reviews related works, and finally, Section 6 concludes the paper and briefly describes future works.

2. DATA FLOW INTEGRITY SPECIFICATION

In this section, we first review Control Flow Graph (CFG) of every program which is a basis of programs Data Flow Graph (DFG), and then utilize these graphs to specify DFI. Definitions of CFG and DFG of programs are extracted from [13].

CFG of a program statically demonstrates all possible sequences in instructions of that program which can be executed at runtime. This graph can be extracted before program execution, e.g. at compile time. To extract CFG, program instructions are divided into basic blocks of instructions. A basic block is a sequence of consecutive statements, not necessarily a maximal sequence in which flow of control can only enter the first statement of the block and leave the last statement without halting or branching except possibly at the last statement in the basic block [13]. Therefore, a program CFG is defined as follows.

Control Flow Graph. The CFG associated with the program is formally defined as a graph $G = (B, E)$ such that B is the set of basic blocks as vertices each of which is identified by a unique number and denoted by b_i. E is the set of edges such that each edge indicates a possible transition of control between two basic blocks and also is identified with a unique number such as $e_m = (b_i, b_j)$ which shows an edge from b_i *to* b_j while b_i, $b_j \in B$. For simplicity, let $b_i \in G$ and $e_m \in G$ respectively denote $b_i \in B$ and $e_m \in E$ of $G = (B, E)$.

Every program CFG has exactly one vertex s as the starting vertex with no incoming edge and exactly one vertex t as the terminating vertex with no outgoing edge. A sequence of vertices and edges $p_i = (s, e_{i_1}, b_{i_1}, e_{i_2}, b_{i_2}, ..., b_{i_{l-1}}, e_{i_l}, t)$ in G is called a program *CFG path* or shortly a *path* such that $e_{i_1} = (s, b_{i_1})$, $e_{i_l} = (b_{i_{l-1}}, t)$, and for all $j \neq 1, l$, $e_{i_j} = (b_{i_{j-1}}, b_{i_j})$ [14][15]. Every program path $p_i = (s, e_{i_1}, b_{i_1}, e_{i_2}, b_{i_2}, ..., b_{i_{l-1}}, e_{i_l}, t)$ shows a sequence of instructions that respectively belong to b_{i_1} through $b_{i_{l-1}}$, with the same order in each $b_{i_j} \in p_i$. Let P be the set of all program paths derived from the program CFG each of them is denoted by p_i while i uniquely identifies each path. At every program execution at runtime, instructions in one of these paths are sequentially executed. For every program execution, such a path is called the program *runtime path.* While CFG and P for each program can be determined statically, runtime path is just determined at runtime.

Program DFG is derived according to semantics of instructions in each program path. Let us mark every instruction that, based on its semantics, defines a particular variable v by $Def(v)$, which means it assigns a value to v, and any instruction that uses that variable by $Use(v)$. In addition, let $x.ML$ for every instruction or block x be the set of all its marks. To be referred uniquely in the analysis, an index is used to identify every instruction mark in the program too. In Figure 1 (a) the CFG basic blocks and edges of a code fragment are respectively shown by rectangles and solid lines, and the ML list for each instruction that legally defines or uses the variable i is shown as its comments. In this figure we assumed *const*, c, c' and c'' as constants.

Let us assume that in the program CFG blocks of instructions are such that each $b_i \in CFG$ exclusively contains at most either one instruction marked as $Def(v)$ or one instruction marked as $Use(v)$ for any particular variable v. This blocking makes the basic block granularity of CFG more suitable that can also be used as the basis of Data Flow Graph.

Date Flow Graph (DFG). Let a DFG of a program with a CFG $G = (B, E)$ be a graph $G' = (B, E')$ such that E' is a set of edges such as $e'_n = (b_i, b_j)$ denoting an edge from b_i to b_j when b_i, $b_j \in B$, and n is a unique identifying number for every element of E', if and only if the following two conditions are satisfied:

1. there exists a variable v in the program such that $Def(v) \in b_i.ML$ and $Use(v) \in b_j.ML$, and
2. there exists at least a path $p_i = (s, \ldots, b_i \ldots, b_j, \ldots, t) \in P$ such that there is not any $b_k \in b_i \ldots, b_j$ of p_i such that $Def(v) \in b_k.ML$,

In such a case, it is said that there is a data flow denoted by $df : Def(v) \rightarrow Use(v) \in DFG$.

According to the above definition, for any $b_i, b_j \in CFG$ of a program, there is a data flow from b_i to b_j for a particular variable v if there is at least one path $p_i = (s, \ldots, b_i, \ldots, b_j, \ldots, t) \in P$ such that there exists only one definition instruction for v in b_i that assigns it a value that is used at an instruction in b_j without any change in-between. In such a case, there is an edge e' from b_i to b_j in the program DFG. Moreover, we call e' an *expected data flow* in all such paths, e.g. p_i. As an instance, in Figure 1 (a) DFG edges of the code fragment are simply shown by dashed lines. According to this figure, in the path $p = (\ldots, b_1, e_1, b_2, e_3, b_3, e_4, b_2, \ldots)$ there are two expected data flows: one of them is from b_1 to the first occurrence of b_2 in p, and the other is from b_3 to the second occurrence of b_2. In addition, a data flow from b_1 to the second occurrence of b_2 is not expected in p.

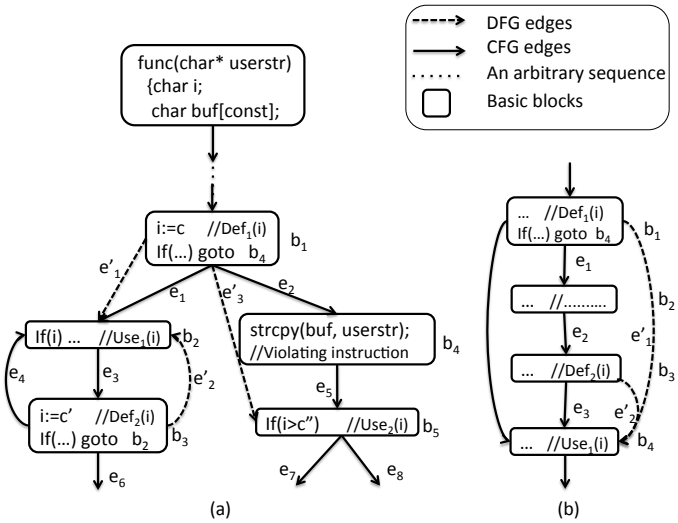

FIGURE 1. Two parts of CFG and DFG of a code fragment

DFG of a program can be statically extracted at compile time using a compiler analysis technique called "Data flow analysis" which refers to a body of techniques that derive information about the flow of data along program execution paths [13].

Definition 2.1 (Runtime Data Flow). *For each runtime path* $RP = (s,\ldots,b_i,\ldots,b_j,\ldots,t) \in P$, *there is a runtime data flow from* b_i *to* b_j *denoted by* $rdf : Def(v) \rightarrow Use(v)$ *for the variable* v *if it assigned a value in* b_i *and its value is used in* b_j *without any change.*

Attacks against data flow integrity can cause runtime data flows that are not allowed by program DFG or are not expected in a specific runtime path. Therefore, we define true runtime data flow or shortly *true data flow* as follows.

Definition 2.2 (True Data Flow (TDF)). *A runtime data flow,* rdf, *from* b_i *to* b_j *for a variable* v *in a runtime path* $RP = (s,\ldots,b_i,\ldots,b_j,\ldots,t) \in P$, *is a true data flow if and only if the next two conditions are satisfied:*

1. rdf *is an* ***existing data flow*** *of the program DFG. In other words, for* b_i *and* b_j *in* RP, *when* b_i *precedes* b_j, *there exists* $Def(v) \in b_i.ML$ *and* $Use(v) \in b_j.ML$ *which means there is an instruction in* b_i *and another in* b_j *that legally define and use the variable according to their semantics, respectively.*
2. rdf *is an* ***expected data flow*** *in* RP *which means there is no other legal definition instruction in* RP *in any* $b_k \in RP$ *in the way from* b_i *to* b_j *in* RP.

Definition 2.3 (Data Flow Integrity (DFI)). *The Data flow Integrity (DFI) is a security policy that dictates every runtime data flow of a program execution should be a true data flow of that program.*

Let us assume an execution of a code fragment in Figure 1 (a) while $p = (...,b_1,e_2,b_4,e_5,b_5,...)$ is its runtime path. Moreover, we assumed an instruction in block b_4 that might be exploited by attacks at runtime, e.g. an instruction that is vulnerable to overflows or out-of-bound reads. For instance, assume that there is an input parameter which is a user supplied data, $userstr$ and an array, buf which in addition to i is a local variable. Usually, these two local variables are placed next to each other on the stack such that i is pushed on the stack and then buf while the stack grows down. Assume that the violating instruction is `strcpy` which has a buffer overflow vulnerability. `Strcpy` copies the user supplied data to buffer even if

$$\frac{}{DFG, rp = () \vdash TDFS = \emptyset} \quad \text{(init)}$$

$$\frac{\begin{array}{c} v \in \vartheta, b \in CFG \vdash Def_n(v) \in ML(b) \\ Def_n(v), DFG \vdash exp_{df} = \{df \in DFG | df = Def_n(v) \rightarrow Use_m(v)\} \\ Def_n(v), TDFS \vdash unexp_{df} = \{df \in TDFS | df = Def_k(v) \rightarrow Use_{k'}(v), n \neq k\} \end{array}}{rp, b, v, ra_b = rdef(v), TDFS \vdash Extend(rp, b), TDFS = (TDFS - unexp_{df}) \cup exp_{df}} \quad \text{(def)}$$

$$\frac{\begin{array}{c} v \in \vartheta, b \in CFG \vdash Use_n(v) \in ML(b) \\ Use_n(v), TDFS \vdash tdfs = \{tdf \in TDFS | tdf : Def_m(v) \rightarrow Use_n(v)\}, tdfs \neq \emptyset \end{array}}{rp, b, v, ra_b = ruse(v), TDFS \vdash Extend(rp, b)} \quad \text{(use)}$$

$$\frac{v \in \vartheta, b \in CFG \vdash Def_n(v) \notin ML(b)}{rp, b, v, ra_b = rdef(v), TDFS \vdash Alarm(Illegal\ \ def)} \quad \text{(ill-def)}$$

$$\frac{v \in \vartheta, b \in CFG \vdash Use_n(v) \notin ML(b)}{rp, b, v, ra_b = ruse(v), TDFS \vdash Alarm(Illegal\ \ use)} \quad \text{(ill-use)}$$

$$\frac{\begin{array}{c} v \in \vartheta, b \in CFG \vdash Def_n(v) \notin ML(b) \\ v, CFG, TDFS \vdash trdf = \{tdf \in TDFS | tdf : Def_k(v) \rightarrow Use_{k'}(v)\} \\ trdf, CFG \vdash ppwrdf = \{p \in CFG | p = (..., b_{Def_k(v)}, ..., b, ..., b_{Use_{k'}(v)}, ...)\} \\ ppwrdf, CFG \vdash pwrdf = \{p \in ppwrdf | b_{Use_{k'}(v)} = PosDom(b)\} \end{array}}{\begin{array}{c} rp, b, v, ra_b = rdef(v), TDFS \vdash Alarm(Illegal\ \ def), \\ wrdf = \{rdf | rdf = (rdef(v), ..., Use_{k'}(v))\} \\ wrdf \neq \emptyset \vdash Alarm(DFIviolation) \end{array}} \quad \text{(def-notTDF)}$$

$$\frac{\begin{array}{c} v \in \vartheta, b \in CFG \vdash Use_n(v) \notin ML(b) \\ v, CFG, TDFS \vdash trdf = \{tdf \in TDFS | tdf : Def_k(v) \rightarrow Use_{k'}(v)\} \end{array}}{\begin{array}{c} rp, b, v, ra_b = ruse(v), TDFS \vdash Alarm(Illegal\ \ use), \\ wrdf = \{rdf | rdf = (Def_k(v), ..., ruse(v))\} \\ wrdf \neq \emptyset \vdash Alarm(DFIviolation) \end{array}} \quad \text{(use-notTDF)}$$

FIGURE 2. DFI rules

it exceeds the buffer size. In such a case, extra data result in i definition while the instruction is not supposed to do that. Therefore, by exploiting this vulnerability in a runtime attack, a runtime data flow from b_4 to b_5 for variable i occurs. Moreover, an attack may perform an out-of-bound read in another violating instruction, such as the example in Figure 6, for the same variable. According to Definition 2.2, non of these runtime data flows are true data flows. It is because, in the former, the violating instruction illegally defines i and in the later, this instruction illegally uses i, while according to semantics of the instruction in b_4, the instruction is not supposed to manipulate i, and thus there is not such a data flow in DFG of the code fragment.

3. DFI DYNAMIC RULES AND APPLICATIONS

Inductive definitions are essential tools in the study of programming languages, including specification and analysis. Utilizing these tools, we develop a set of DFI rules that should be satisfied dynamically at runtime. Then, we provide some synthetic and real-world examples of applications and violations of these rules. As we explain in Section 4, every DFI enforcement technique should guarantee satisfaction of these rules at each step of program execution.

3.1. DFI Dynamic Rules

Assume that CFG and DFG are statically derived control flow and data flow graphs of a given program, and rp at each point of the program execution shows the sequence of executed instructions till that point. Moreover, assume that $TDFS$ at each point of the program execution is a set of all possible true data flows that according to DFG and executed instructions till that point can occur during the rest of program execution. With these assumptions, DFI rules in Figure 2 show how rp is continuously going to be completed step by step while DFI is satisfied. Furthermore, these rules show how $TDFS$ is dynamically affected based on statically derived DFG and executed instructions in rp till each point of the program execution at runtime, and all these rules aim to make sure about the truth of occurring runtime data flows.

A statement about one or more of these objects forms a basic judgment in DFI rules. For a given static analysis, data flow analysis, we define the *hypothetical judgement*, written $J_1 ... J_n \vdash J$, where each J_i and J are basic judgements, to mean that according to the static analysis we can derive J from $J_1, ..., J_k$. Furthermore, if for a given set, R, of rules we can derive J from $J_1, ..., J_k$, we have hypothetical judgement written as $J_1, ..., J_n \vdash_R J$. Moreover, an inductive definition of a judgement form consists of a collection of inference rules of the form $\frac{J_1,...,J_k}{J}$ in which J and $J_1, ..., J_k$ are all judgements of the form which was being defined. Judgements above the horizontal line in an inference rules are called the *premises* of the rule, and judgements below the line are called its *conclusions*. If a rule has no premises, it is called an *axiom*.

According to Figure 2, (init) is an axiom that shows the initial value for $TDFS$ at the start of the program execution. It states that by the given DFG of a program and empty sequence rp, $TDFS$ is an empty set. Considering a variable v in the set of all program variables, ϑ, when the next execution step is executing instructions in block b of the program CFG, and the runtime action in b is a definition or use for v, denoted by $ra_b = rdef(v)$ or $ra_b = ruse(v)$, then other rules in Figure 2 are applied and check the truth of data flows at runtime.

According to the second rule, (def), if the runtime action in b is a definition for v, i.e. $ra_b = rdef(v)$, and according to semantics of instructions in b, v is supposed to be defined, i.e. there is a $Def_n(v) \in ML(b)$, then the program execution will be continued by extending rp with execution of instructions in b, shown by $Extend(rp, b)$, while it does not cause any DFI violation. Furthermore, by this new definition, which its instruction is included in rp, all existing data flows in $TDFS$ that define and use the same variable are not expected anymore because the variable is just redefined. Instead, a set of new data flows that start from this new definition are expected. Therefore, these two sets of data flows are collected in $unexp_{df}$ and exp_{df} sets while the $TDFS$ set will be updated by eliminating and adding data flows in $unexp_{df}$ and exp_{df} sets, respectively.

According to the third rule, (use), when the runtime action in b is a use for v, i.e. $ra_b = ruse(v)$, then rp will be extended by instructions in b if there is a legal use of v in b, i.e. $Use_n \in ML(b)$, and there is a definition for the same variable proceeding this use such that they make a true data flow for this variable at runtime. It means, by the given $Use_n(v)$ a true data flow $tdf : Def_m(v) \rightarrow Use_n(v)$ must be found in $TDFS$. It is because we assume that at each runtime path, every variable should be defined before any uses. Similarly, there is no DFI violation in this case.

The next two rules, i.e. (ill-def) and (ill-use), simply explain illegal definition and use of a variable, and will be completed in the last following rules. Rule (ill-def) says that, if the runtime action in b is a definition for v, i.e. $ra_b = rdef(v)$, and according to semantics of instructions in b, v is not supposed to be define, i.e. there is not any $Def_n(v) \in ML(b)$, then it is an error and an alarm will be raised about an illegal definition. Similarly, (ill-use) says that, if the runtime action in b is a use for v, i.e. $ra_b = ruse(v)$, and according to semantics of instructions in b, it is not allowed, i.e. there is not any $Use_n(v) \in ML(b)$, then an alarm about an illegal use will be raised. In both of these cases, continuing the program execution by

$$\frac{\begin{array}{c} i \in \vartheta, b_1 \in CFG \vdash Def_1(i) \in ML(b) \\ Def_1(i), DFG \vdash exp_{df} = \{Def_1(i) \rightarrow Use_1(i), Def_1(i) \rightarrow Use_2(i)\} \\ Def_1(i), TDFS \vdash unexp_{df} = \emptyset \end{array}}{\begin{array}{c} rp = (...), b_1, i, ra_{b_1} = rdef(i), TDFS \vdash rp = (..., b_1), \\ TDFS = \{...., Def_1(i) \rightarrow Use_1(i), Def_1(i) \rightarrow Use_2(i)\} \end{array}} \quad \text{(def-example)}$$

$$\frac{\begin{array}{c} i \in \vartheta, b_4 \in CFG \vdash Def_n(v) \notin ML(b) \\ i, CFG, TDFS \vdash trdf = \{Def_1(i) \rightarrow Use_1(i), Def_1(i) \rightarrow Use_2(i)\} \\ trdf, CFG \vdash ppwrdf = \{(..., b_1, ..., b_4, ..., b_5, ...)\} \\ ppwrdf, CFG \vdash pwrdf = \{(..., b_1, ..., b_4, ..., b_5, ...)\} \end{array}}{\begin{array}{c} rp = (..., b_1), b_4, i, ra_{b_4} = rdef(i), TDFS \vdash Alarm(Illegal\ \ def), \\ wrdf = \{rdef(i), ..., Use_2(i)\} \\ |wrdf| \neq \emptyset \vdash Alarm(DFIviolation) \end{array}}$$

(def-notTDF-example)

$$\frac{\begin{array}{c} i \in \vartheta, b_4 \in CFG \vdash Use_n(i) \notin ML(b_4) \\ i, CFG, TDFS \vdash trdf = \{Def_1(i) \rightarrow Use_1(i), Def_1(i) \rightarrow Use_2(i))\} \end{array}}{\begin{array}{c} rp = (..., b_1), b_4, i, ra_{b_4} = ruse(i), TDFS \vdash Alarm(Illegal\ \ use), \\ wrdf = \{rdf | rdf = (Def_1(i), ..., ruse(i))\} \\ |wrdf| \neq \emptyset \vdash Alarm(DFIviolation) \end{array}}$$

(use-notTDF-example)

$$\frac{\begin{array}{c} i \in \vartheta, b_2 \in CFG \vdash Use_1(i) \in ML(b_2) \\ Use_1(i), TDFS \vdash tdfs = \{Def_1(i) \rightarrow Use_1(i)\}, tdfs \neq \emptyset \end{array}}{rp = (..., b_1), b_2, i, ra_{b_2} = ruse(i), TDFS \vdash rp = (..., b_1, b_2)} \quad \text{(use-example)}$$

$$\frac{\begin{array}{c} i \in \vartheta, b_3 \in CFG \vdash Def_2(i) \in ML(b_3) \\ Def_2(i), DFG \vdash exp_{df} = \{Def_2(i) \rightarrow Use_1(i)\} \\ Def_2(i), TDFS \vdash unexp_{df} = \{Def_1(i) \rightarrow Use_1(i), Def_1(i) \rightarrow Use_2(i))\} \end{array}}{\begin{array}{c} rp = (..., b_1, b_2), b_3, i, ra_{b_3} = rdef(i), TDFS \vdash TDFS = \{Def_2(i) \rightarrow Use_1(i)\}, \\ rp = (..., b_1, b_2, b_3) \end{array}}$$

(def-example)

FIGURE 3. Examples of DFI rule applications

instructions in b may create data flows that are not true.

Rule (def-notTDF) completes the situation of an illegal definition which is stated by (ill-def). According to this rule, in case of an illegal definition for a variable, e.g. v, the set of data follows in $TDFS$ for the same variable, i.e. $Def_k(v) \rightarrow Use_{k'}(v)$, is collected in $trdf$. Then, paths which contain one of these data flows while b, and thus its current action, locates in between their definition block, $b_{Def_k(v)}$, and their use blocks, $b_{Use_{k'}(v)}$, are collected in $ppwrdf$. This set consists of possible paths that may contain wrong data flows, from $rdef(v)$ to $Use_{k'}(v)$, that program execution may follow. Further, if there is a block such as $b_{Use_{k'}(v)}$ which postdominate block b, in each path of $ppwrdf$ containing these two blocks, wrong data flows will occur. It is because postdominating b by $b_{Use_{k'}(v)}$ guarantees that every path from b to the end of the program execution, i.e. the exit of the graph, goes through $b_{Use_{k'}(v)}$. This subset of paths is collected in $pwrdf$. Finally, $wrdf$ gives the set of wrong data flows that would be generated because of the illegal definition by the runtime action and a predictive alarm is raised if $wrdf$ is nonempty. In this case, attacks intend to substitute true data flows in $trdf$ by wrong data flows in $wrdf$.

Similar to (def-notTDF), the last rule, (use-notTDF), completes the situation of an illegal use for a variable that is stated by (ill-def). According to this rule, if in addition to an illegal use for a variable, there is also a set of data follows in $TDFS$ for the same variable, denoted by $trdf$, then a set of wrong data flows with the set

of paths containing them are achieved in *wrdf* and *pwrdf* sets, respectively. In this case, an additional alarm is raised that informs about a violation of DFI if *wrdf* is a nonempty set. All these DFI rules explain the situation when the runtime action in b is a variable manipulation. Otherwise, rp will be just extended by b.

3.2. Examples of DFI Rule Applications

Examples of DFI rule applications and violations are as many as memory vulnerabilities and exploits themselves. Moreover, since DFI violation constitute a major step in many other complicated security attacks including but not limited to denial of service and attacks against confidentiality, to describe examples of DFI rules, we aim to provide some synthetic and a few live exploits with original well-known vulnerabilities as simple as possible.

To see synthetic examples, lets get back to Figure 1 (a). Assume an alternative sequence of executed instructions till entrance of b_1 in rp denoted by $rp = (...)$ while runtime action is a definition for variable i, i.e. $ra_{b_1} = rdef(i)$. Since b_1 can legally define this variable, it has a definition mark $Def_1(i)$. In this case, rule (def) can be applied which is shown in (def-example) of Figure 3. Also, we assume that the definition in b_1 is the first definition for i in rp, and thus the $unexp_{df}$ set is an empty set. Therefore, according to rule (def), the runtime action does not cause a wrong data flow, and $exp_{df} = \{Def_1(i) \rightarrow Use_1(i), Def_1(i) \rightarrow Use_2(i)\}$ will be added to the existing set of $TDFS$, and rp will be extended to $(..., b_1)$.

If the execution proceeds while the next step is running instructions in b_4, $ra_{b_4} = rdef(i)$, and there is not any $Def(i)$ in $ML(b_4)$, then rule (def-notTDF) can be applied as shown in (def-notTDF-example). In such a case, an alarm is raised that informs about an illegal definition. Moreover, a set of true data flows for variable i that are expected to occur is find in $tdfs$, and are used to find the set of wrong data flows in $wrdf$. Wrong data flows are formed of $rdef(i)$ in b_4, and instructions that use i which will be followed in the continue of the program execution. Since $b_5 = PosDom(b_4)$, in every program execution that instructions in b_4 are executed, then execution of instructions in b_5 will be followed. Therefore, existence of a wrong data flow from $rdef(i)$ to $Use_2(i)$, and thus DFI violation are predicted and an alarm is raised. Similarly, if the runtime action in b_4 is $ra_{b_4} = ruse(i)$, then (use-notTDF-example) shows the application of (use-notTDF). In such a case, a set of wrong data flows would be $wrdf = \{Def_1(i) \rightarrow ruse\}$, and thus a DFI violation is detected in addition to raising an alarm about an illegal use. Other examples in (use-example) and (def-example) show a legal usage and definition in b_2 and b_3, respectively. These two examples, that are easy to follow, show applications of (use) and (def).

Now after explaining DFI rules, we present some simple examples from many real-world software applications that are vulnerable to attacks against DFI such as Wu-FTP and SSH server [2], which are the most widely used FTP and one of widely used SSH servers, respectively. In both of these cases (def-notTDF-example) of DFI rules are violated, i.e. an illegal definition happens, while in the former the DFI violation may happen if an specific path is followed at runtime, and in the later the DFI violation is inevitable. For the sake of simplicity in these examples, we ignore showing CFG or DFG of the code fragments which can be extracted from the code. Moreover, to prevent unnecessary complexities and replication of vulnerability explanations, we ignored explaining these examples with more details.

Wu-FTP contains a format string vulnerability allowing attackers to violate DFI by performing illegal definition of an arbitrary memory location. In the violating code fragment shown in Figure 4, there is an integer x declared to store a copy of the user ID of the authenticated user. By default, at the start of this code, the effective user ID, EUID is set to 0 indicating the root privilege. When the user is authenticated, x will be assigned the user ID. Assume a regular user representing by x=n defined in line 4 while EUID still indicates the root privilege. Therefore, it is needed to lose the root privilege by calling seteuid(x) and changing EUID to n by coping x to EUID. When the program gets to line 7 a special site

```
1:  int x;
2:  FTP_service(...) {
3:  authenticate();                                  // x is uninitialized and EUID is 0
4:  x = user ID of the authenticated user;                    // x=n and EUID is 0
5:  seteuid(x);        // Lose the root privilege, i.e. copy x to EUID, x is n and
    EUID=n.
6:  while (1) {
7:    get_FTP_command(...);        // Exploit a format string vulnerability and
    define x=0 while Def(x) ∉ ML(b) and EUID is n.
8:    ...
9:    if (a data command)
10:     getdatasock(...);
11:   }
12: }
13:   ...
14: getdatasock( ... ) {
15:   ...
16:   seteuid(0);                                                  // x is 0, EUID=0
17:   setsockopt( ... );
18:   ...
19:   seteuid(x);   // use(x) ∈ ML(b), again wants to lose the root privilege by
    copying x to EUID, but x is 0 and thus EUID=0 still indicating root privilege.
20:   ...
21: }
```

FIGURE 4. An example of DFI attack in an FTP server.

exec command exploits the format string vulnerability in get_FTP_command to overwrite x to 0 while EUID is n, while according to the its semantic, it is not supposed to. Moreover, when execution of this application reaches invoking the function getdatasock, the privilege of the process needs to be temporarily elevated by defining EUID=0 to make this system call. Before this function returns, it tries to drop the privilege by seteuid(x). Since x has been overwritten to 0, the process does not drop the privilege. So, when the function returns to the service loop, the program runs as root. Therefore, by exploiting a vulnerability and performing an illegal definition, attackers can grant themselves the root privilege and do their malicious valuable desires including uploading any file such as /etc/passwd.

There is an integer overflow vulnerability in widely used SSH server allowing attackers to overwrite a memory location which is shown in Figure 5. In this code fragment, function packet_read has an integer overflow vulnerability and function do_authentication() authenticates remote users. A variable auth is defined indicating whether the user has succeeded in the authentication. The initial value of auth is 0. While the user is not authenticated, i.e. in *while* loop, the vulnerable function is called to read the password. At this point, an attacker can overwrite the auth to 1. When the program tries to verify the password, it fails, but it does not matter because auth, which is legally supposed to be 0, has already been set to 1. Finally, the server lets the attacker to log in although the correct password is not known.

Similarly, there are many applications with vulnerabilities which can be exploited to perform DFI attacks through violating use-notTDF-example. For example, isakmpd packets containing a vulnerability in TCPDUMP 3.8.1 [16], and in isakmpd of OpenBSD 3.4 [17], which cause an out-of-bound read. In these cases, a memory location is read which is not legally supposed to and thus causes an illegal use of a memory location. A simpler code fragment is shown as an example in figure 6. This code accesses values of an array elements but it just verifies that the given array index is less than the maximum length of the array. However, it does not check for the minimum value of array index which will result in an out-of-bound read and

```
void do_authentication(char *user, ...) {
int auth = 0;
...
while (!auth) {
 type = packet_read();                          // Def(auth) ∉ ML(b)
 switch (type) {
    ...
    if (auth_password(user, password))
      auth =1;
    case ...
  }
 if (auth) break;                               // Use(auth) ∈ ML(b)
}
 ...
}
```

FIGURE 5. An example of DFI attack in an SSH server.

```
1:  int getArrayValue(int y, int *array, int len, int index) {  // Def(y) ∈ ML
2:   ...
3:  int value;
4:  ...
5:  if (index < len) {
6:    ...
7:    value = array[index];                     // use(y) ∉ ML
8:  }
9:   else
10:   ...
11: }
```

FIGURE 6. An example of DFI attack caused by an illegal use.

may allow access to sensitive memory. In this example, we assumed a variable y which is stored right before the array and thus the out-of-bound read causes y to be used. This variable is defined in line 1, and thus in addition to an illegal use, a wrong data flow will be generated.

4. DFI ENFORCEMENT

Since security attacks exploit vulnerabilities at runtime to violate DFI, the enforcement of this policy should dynamically monitor program behavior at runtime. Therefore, we provide two techniques based on two well-known enforcement mechanisms, Reference Monitors (RMs) [10, 11] and Software Memory Access Control (SMAC) [12], and some security properties including Control Flow Sequentiality (CFS). DFI enforcement has two different aspects: security-related specification and guarantee from an abstract point of view, and platform-related implementation. In this paper, our main focus is on security-related features and DFI specification. We believe that for any platforms and languages with various supported features, a suitable implementation with reasonable overhead can be provided. However, we measured the enforcement overhead of our approach for a sample program to present an example of a worst-case overhead.

4.1. Reference Monitors (RMs)

Reference monitors that encompass most current security enforcement mechanisms at runtime, observe software execution for a given security policy, and if an operation violates the policy, they take remedial actions [11]. There are several

kinds of reference monitors. Traditionally, reference monitors reside in OS kernels and before certain machine instructions halt program execution and check satisfaction of desired security policy. In an alternative implementation, programs are executed inside an interpreter and a reference monitor is called before each instruction. An inlined reference monitor (IRM) is another alternative that obtained by modifying program code to include the policy enforcement code of a reference monitor. The last alternative is more flexible than traditional RMs for enforcing policies. After applying an RM, a summary of the target execution history, as relevant to the desired policy, is maintained in a protected part and satisfaction of the security policy is dynamically checked. Moreover, in case of a violation detection a remedial action is taken by the resulted code. As a result of applying the enforcement mechanism, the target software is called secured. Furthermore, effective RM enforcers should meet the requirements below.

- Soundness: observable executions of secured application by RM must satisfy the desired policy.
- Transparency: observable behavior of the program in the absence of security policy violations must not be changed.
- Integrity: circumvention of RM enforcement code must be prevented.

4.2. Software Memory Access Control (SMAC)

Software Fault Isolation (SFI) is an enforcement mechanism that with the aim of memory protection performs dynamic checks [18]. SMAC is an extension of SFI in which access checks are inserted at different instructions in the program to impose access memory restrictions. In particular, SMAC can create isolated memory regions that are accessible from only specific pieces of program code, for instance, from individual instructions in IRM enforcement code.

4.3. Control Flow Sequentiality

CFS is a security property that dictates the sequence of instructions at runtime must be according to a path of the program CFG. CFS is a necessary condition of CFI which is a defined security policy in [12]. In general, CFS is related to security policies derived from analysis of programs code or their executions, and has cooperation with some enforcement mechanisms such as IRM and SFI [12]. In addition, satisfaction of this property has a major affect on DFI. If the sequence of blocks can be subverted in a program execution, then it is possible for program execution to reach a block with a legal runtime action that just is not expected. Such problems cause unexpected and thus wrong runtime data flows in program executions. In specification of DFI and developments of its rules in Section 3, we consider CFS satisfaction. By this assumption, execution of a sequence of instructions in a program that does not correspond to any CFG path of that program is impossible. For example, consider Figure 1 (b) when CFS is not satisfied, and thus a runtime path could be such as $(..., b_1, b_2, b_4, ...)$ which does not correspond to any CFG path. In such a case, a runtime data flow can occur from Def_1 to Use_1 which although exists in the program DFG, is not a true data flow because there is not any path such as $(..., b_1, b_2, b_4, ...)$ that such a data flow is expected in it. Moreover, post dominants rule used in DFI rules entails CFS satisfaction which means if a block b postdominates block b', then every runtime path from b' to the exit of the graph goes through b if CFS is satisfied. Since CFS satisfaction prevents illegal jumps in the program execution, in addition to its necessity for DFI enforcement, it prevents circumvention of enforcement code especially in IRM.

4.4. Enforcement techniques

To enforce DFI, it is needed to propose techniques that dynamically monitor program behavior at runtime. According to DFI specification and rules, it is

required to do some static operations such as statically extracting program CFG and DFG, and to provide suitable situations to maintain them such as in a protected part and providing a model of program execution to monitor its behavior to keep track runtime data flows.

CFG and DFG of a program can be extracted before its execution, e.g. at compile time. Program CFG is derived according to basic blocks and instruction semantics related to transition of control flow between instructions which is described in [13]. Moreover, according to semantics of every instruction, a mark list that determines which data manipulations are legal by each instruction is derived. Then, utilizing some data flow analysis such as reaching definitions that is one of the most common and useful ones, and point-to which is popular in pointer analysis, program DFG can be extracted. Reaching definition with other related analysis such as points-to analysis are described in [13] and [19] in details which are also improved in some other works such as [20].

In the enforcement techniques, it is required to have a protected part to save some information about program behavior. For example, information about program CFG and DFG that are derived statically and used dynamically at runtime should be maintained in the protected part. Moreover, it is needed to keep track of true runtime data flows, i.e. $TDFS$, dynamically at runtime just by enforcement code. The requirement of having a protected part for runtime policy enforcement especially in program monitoring is a common requirement in lots of works which is provided based on software, e.g. similar strategies to SMAC used in [1, 7], and/or hardware, e.g. a machine mode used in [21].

With the aim of analysis and monitoring, execution of instructions along a program runtime path can be viewed as program state transitions [11, 13]. Each program state may consist of information based on program data memory, stack, registers, code memory and etc. For every instruction, the associated state with the program point before the instruction and the associated state with the program point after the instruction are respectively called the instruction *input state* and the instruction *output state* [13]. The execution of each instruction transforms this input state into the output state. In this paper, all program states consist of two different parts, a fixed part that is constructed before execution and will not be changed at runtime, and an unfixed part that will be modified at runtime. The former consists of information about program CFG, DFG and code memory M_c, and the later consists of $TDFS$, program counter pc, information about actual ongoing runtime path rp, and the next runtime action, i.e. ra in block b, that might be included in rp as the summery of program execution. When s_i is the state of a specific program point, we may write $s_i.pc$ to refer to the pc component of s_i when the other components can be referred similarly. Data flow dynamic rules are satisfied if unfixed components of states are changed according to them.

Without addressing exactly how RM is implemented to check satisfaction of the desired policy, detect violations and take remedial actions, we provide a main theorem that shows what assumptions and guarantees must be expected of a DFI enforcement technique based on RM.

Let $s_0, ..., s_n$ be states of a program execution such that $\sigma = s_0 \rightarrow s_1 \rightarrow ... \rightarrow s_n$ denotes the sequence of state transition in a runtime path while s_i, $0 \leq i \leq n$ has the form of $(CFG, DFG, Mc = RM(program), TDFS, rp, ra, b, v, pc)$, when $RM(program)$ denotes the code memory of the program after applying the enforcement mechanism. Moreover, the initial state is $TDFS = \emptyset, rp = (), ra = non, b = start, v = non, pc = 0$. Actually, inductive definition provided in Figure 2 specifies changes in unfixed state components. For example, every $TDFS_i \vdash TDFS_j$ in conclusions of judgments implies a state transition such as $s_i.TDFS_i \rightarrow s_{i+1}.TDFS_j$, and for simplicity we denote such transitions by $s_i.TDFS_i \vdash s_{i+1}.TDFS_j$. If in an instruction is not manipulating any variable, then its input and output states are the same.

In this situation, in the underlying system of a DFI enforcement technique based on RM, there must be guarantees or at least assumptions as follows:

1. there is a supported protected part denoted by $ProtPart$ as required in the enforcement technique,
2. CFS is satisfied that means the sequence of instructions which are executed is according to a path of the program CFG, i.e. $\forall i, 0 \leq i \leq n$, $s_i.pc, CFG \vdash s_{i+1}.pc = succ(s_i.pc)$ when $succ()$ for every pc returns the set of allowed pc that according to the given CFG are its successors,
3. the program code is Non Writable Code (NWC) which means it is not possible to modify the program code memory at runtime,
4. the program data are Non Executable Data (NXD) which means it is not possible for the program to execute data as if it were code,
5. No Other programs can Change (NOC) the program data memory or its runtime states. Therefore, if $Instruction1$ and $Instruction2$ with state transitions $s_{i-1} \xrightarrow{Instruction1} s_i$ and $s'_i \xrightarrow{Instruction2} s_{i+1}$ are two sequential program instructions, according to NOC, $s_i = s'_i$,

According to these assumptions we establish DFI main theorem which shows effectiveness of the proposed dynamic inference rules to satisfy DFI. Actually, we focus on the assumptions that are most related to the paper concept. For example, we neglect the possible hardware faults which may affect instructions in arbitrary ways [22].

THEOREM 4.1 (DFI main Theorem). *By the above assumptions, and a given precise CFG and DFG of a program, DFI is satisfied if and only if data flow dynamic rules (DDR) are satisfied. Briefly,* $(\sigma = s_0 \rightarrow s_1 \rightarrow ... \rightarrow s_n \wedge ProtPart \wedge CFS \wedge NWC \wedge NXD \wedge NOC) \Rightarrow (\forall i, DFI \Leftrightarrow s_i \vdash_{DDR} s_{i+1})$ *where* $0 \leq i \leq n$.

Proof. To prove $DFI \Rightarrow s_i \vdash_{DDR} s_{i+1}$, we use rule induction. The inductive definition provided in Figure 2 specifies changes in state components that imply the judgement $s_i \vdash_{DDR} s_{i+1}$. Since such judgment is defined using the collection of rules in Figure 2, we can reason about it by rule induction on those rules. The principle of rule induction states that to show a defined property ρ holds of a judgement J whenever J is derivable, it is enough to show that ρ is provable under the rules defining J. To do so, first we define $\rho(J)$ as follows:

1. let $\rho(J : axiom)$ be if DFI holds then axiom holds or simply if DFI then axiom, and
2. $\rho(J : (J_1 : a, TDFS_1 \vdash J_2 : b, TDFS_2))$ be if DFI and a and $s_i.TDFS = TDFS_1$, then b and $s_i.TDFS_1 \vdash s_{i+1}.TDFS = TDFS_2$, and
3. $\rho(J : (J_1 : a, rp_1 \vdash J_2 : b, rp_2 = Extend(rp, b)))$ be if DFI and a and $s_i.rp = rp_1$, then b and $s_i.rp_1 \vdash s_{i+1}.rp_2$, and
4. $\rho(J : (J_1 ... J_n \vdash J))$ be if $\rho(J_1), ..., \rho(J_n)$ then $\rho(J)$.

Therefore, for every J in DFI dynamic rule it is required to prove $\rho(J)$ as follows:

$\rho(J : init)$: It is obvious if DFI, according to Definition 2.3, holds and rp is an empty set then there is not any occurred definition and thus TDFS must be an empty set either. This case explains the initial state $rp = () \vdash s_0.TDFS = \emptyset$ and $s_0.rp$ is not changed.

$\rho(J : def)$: if DFI holds $\wedge$ $(v \in \vartheta, b \in CFG \vdash Def_n(v) \in ML(b)) \wedge$ $(Def_n(v), DFG \vdash exp_{df} = \{df \in DFG | df = Def_n(v) \rightarrow Use_m(v)\}) \wedge$ $(Def_n(v), TDFS \vdash unexp_{df} = \{df \in TDFS | df = Def_k(v) \rightarrow Use_{k'}(v), n \neq k\})$, and all assumptions hold, then according to DFI, unexpected data flows should not be considered as true data flows. In contrast, the set of expected data flows that can be generated by a legal definition for v should be considered as true data flows. Therefore, $s_i.TDFS \vdash s_{i+1}.TDFS = (TDFS - unexp_{df}) \cup exp_{df}$. Moreover, since there is not any violation, then runtime path in s_{i+1} will be extended as stated by $s_i.rp \vdash s_{i+1}.rp = Extend(rp, b)$. Similarly, $\rho(J : use)$ can be proved.

$\rho(J : ill - use)$ and other rules are vacuously true because in those rules the required action contradicts DFI.

We prove the other direction of the theorem, i.e. $s_i \vdash_{DDR} s_{i+1} \Rightarrow DFI$ by contradiction. If DFI is not satisfied, then there is a runtime data flow which is not

an existing or expected data flow. In other words, in the first case, there is a runtime definition in a block b for a variable v, i.e. $ra_b = Def(v)$ while $Def(v) \notin ML(b)$. This situation contradicts def-notTDF. Similarly, assuming $ra_b = Use(v)$ while $Use(v) \notin ML(b)$ contradicts def-notTDF. Moreover, in the second case, we assume that there is a runtime data flow which is not an expected data flow, i.e. although the data flow constructs of a legal definition and use for a variable, it is not expected to be executed in a particular order. For example, in Figure 1(b), if CFS in not satisfied the runtime path might be such as $(..., b_1, b_2, b_4, ...)$ with a runtime data flow from Def_1 to Use_1 which although exists in the program DFG, is not an expected data flow. Clearly, assuming occurrence of such an unexpected data flow at least contradicts to one of DFI assumptions such as CFS. □

It is however critical that described assumptions hold because if they invalidated somehow, security guarantees can be reduced or void. Moreover, since most software features are rather static, we assumed static features which are practical. However, presence of dynamic features does not correspond to our assumptions or policy specification. Examples of dynamic features might be self-modifying code, runtime code generation, and dynamic loading of code. However, we have considered studying effects of these features on DFI as a future work.

One possible technique to implement a protected memory can employ SMAC in which a part of memory that is in a memory region whose addresses is in a specific boundary is protected. It is because SMAC inserted checks only ensure that individual instructions, e.g. instructions in enforcement code, can modify such a memory region. Therefore, it supports isolated memory regions in which the RM summery of the program execution can be safely kept. Moreover, Abadi in [12] shows CFS enforcement utilizing SMAC embodied in an IRM implementation while it can remove the need for NWC, by disallowing writes to code memory addresses, and the need for NXD, by preventing control flow outside code memory addresses. Moreover, SMAC could be circumvented if control flow transfers into or around the code sequences that perform SMACs related checks which CFS prevents such subversion. Hence, CFS and SMAC can greatly cooperate in DFI enforcement and by satisfying NWC and NXD of assumptions make the DFI main theorem simpler.

To evaluate overhead of the enforcement, it is worth noting that our main aim in this paper is not contributing an implementation with optimized overhead to enforce DFI because different architectures, and characteristics of applications with different services may affect the optimized implementation. Moreover, since different techniques, such as those presented in related works, are applied on different platforms and different sets of programs against their intended attacks, it is hard to precisely compare them based on their imposed overhead. However, to perform an overhead measurement of the DFI enforcement at least for an appropriate case, we chose a common program of the integer benchmarks in SPEC 2000 [23] which is a mostly used benchmark suite for evaluation. Among the programs in this benchmark, we selected crafty which has the highest overhead in the most related approaches such as [1]. Moreover, with the same way used in [1], we extract DFG according to the analysis provided in [13]. Then, to satisfy DFI dynamic rules and according to them, we instrument every instructions that may define and use a memory in the program. Similarly, we provide required protected part by instrumenting writes to ensure the target address is not within the memory region allocated to the protected part. The normalized overhead of crafty for both of two variants of enforcement in [1] when no optimization is applied is 2.3 while its space overhead is 0.6. The normalized overhead is defined as the ratio of the overhead of execution time of the program after enforcement to the execution time of the original program in the benchmark [1]. The worst-case normalized overhead of our enforcement for crafty is 2.7 while its space overhead is 0.7. The worst-case overhead happens when no violation is detected and thus the program runs normally with all checks. Due to considering a more complete DFI specification and thus performing more operations in the enforcement, the extra overhead without any optimization is reasonable. Since the SPEC benchmarks focus on CPU intensive

programs with integer arithmetic, we believe that the DFI enforcement results in lower overhead for programs that are not CPU intensive like I/O intensive programs or those that spend a long time in kernel. In general, dynamic approaches are widely used to provide security features especially in critical applications. Sometimes more attempts are made to optimize an existing approach. Hence, there always should be a trade off between the aim of dynamic approaches and their imposed overhead. However, we leave proposing an optimized implementation for DFI enforcement as our future work.

5. RELATED WORKS

Direct or indirect attention to DFI is not new. Some related works focus on preserving data integrity or, more general, memory safety, e.g. [24, 25, 5, 26, 27, 7]. Some others, often are static approaches, and mostly do not consider real situations of powerful runtime attacks, e.g. [26, 27]. Furthermore, the most related approaches in addition to have undetected violations, often made incomplete and informal specification of the policy with unclear assumptions and do not establish formal guarantees, e.g. [1, 7].

Experiences shows that attackers more intend to make an unauthorized change to a data to abuse it in the continuation of the program execution or make an unauthorized use of a data that is not intended by the programmer. Therefore, violations of data flow integrity are more desired in program attacks than integrity of data. Moreover, although some techniques indirectly impose data flow restrictions by satisfying data integrity, they focus on some specific critical data protections and do not detect all violations of data flow integrity. For example, there are many techniques that check data integrity stored in a specific memory such as stack and defend from unexpected data flow to return addresses, e.g. [24, 25].

Memory safety enforcement is another helpful technique to preserve data integrity which for example is used in CRED [5], CCured [26] and Cyclone [27]. Although bound checkers such as CRED can detect some data attacks, they do not prevent all bound violations such as attacks that exploit format string vulnerabilities or those that overwrite data by using pointers. Moreover, CCured comes with a heavy wrapper and impose significant and non-trivial changes to codes by changing instructions instead of monitoring and controlling behavior of existing code at runtime. For example, in both of CCured and Cyclone a garbage collector is used as a substitute for malloc and free in C.

DFI is defined in [1] for the first time, and is enforced by an approach that makes many exploits unsuccessful. However, the informal definition of DFI, which states *whenever a value is read, the definition identifier of the instruction that wrote the value should be in the set of reaching definitions for the read*, mostly relies on its specific enforcement and implementation. Moreover, every definition and use are identified, and then program DFG is computed using static analysis. Further, the enforcement approach instruments the program to dynamically ensure its defined DFI. This approach has false negative, for example due to using imprecise analysis to extract the program DFG which does not distinguish between fields of structures. In case of possibility of CFS violation or code memory modification, which are not clearly discussed, this rate of false negatives will be increased. Moreover, the definition and thus the enforcement do not cover all kinds of data flow integrity violations. Briefly, according to DFI definition in [1], whenever a value is read something is checked about the value, but it is not checked if the read is authorized. Therefore, although runtime data flows that are generated by unintended data usage are not allowed by DFG, e.g. data flows that according to Rule (use-notTDF) are wrong, this approach does not detect them. The memory safety approach presented in Write Integrity Testing (WIT) [7] is another useful technique that uses a static data flow analysis and generates instrumented code to partially prevent instructions from modifying objects that according the static analysis are not allowed to be modified. Although similar to [1], it is a close work to ours because of dynamic checks based on static analysis, it also has the same drawbacks such as incomplete

specification and enforcement for DFI with false negatives.

6. CONCLUSION

The absence of DFI specification and its runtime satisfaction enables many of current security attacks to achieve their final goals or complete an important step in more complicated attacks. Lack of clear assumptions, explicit and independent specification and expectation of the policy satisfaction results in enforcement techniques which not only does not cover all cases of DFI violations but also has false positive and/or false negative, and can be bypassed by powerful attackers.

In this paper, we accomplish a formal study of DFI and develop a precise formulation of hypotheses, guarantees, and proofs in specification and enforcement of DFI which show the major difference between our study and other related works, and an important similarity with research on type-safe languages.

We believe that DFI has major affects on some other security policies. Moreover, enforcement of this policy as a built-in feature in well-known and practical languages with other sophisticated features is very useful. We consider both of these works as our future works.

REFERENCES

[1] Castro, M., Costa, M., and Harris, T. (2006) Securing software by enforcing data-flow integrity. *Proceedings of the 7th symposium on Operating systems design and implementation* OSDI '06, pp. 147–160. USENIX Association.

[2] Chen, S., Xu, J., Sezer, E. C., Gauriar, P., and Iyer, R. K. (2005) Non-control-data attacks are realistic threats. *Proceedings of the14th USENIX Security Symposium.*

[3] National vulnerability database statistics. http://nvd.nist.gov/statistics.cfm.

[4] S.M. Christey, J. K., R.P. Glenn and et al. (2013). Common weakness enumeration (cwe version 2.5).

[5] Ruwase, O. and Lam, M. S. (2004) A Practical Dynamic Buffer Overflow Detector. *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pp. 159–169. Citeseer.

[6] Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G., Frantzen, M., and Lokier, J. (2001) Formatguard: Automatic protection from printf format string vulnerabilities. *Proceedings of the 10th conference on USENIX Security Symposium.* USENIX Association.

[7] Akritidis, P., Cadar, C., Costa, M., and Castro, M. (2008) Preventing Memory Error Exploits with WIT. *Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP '08)*, pp. 263–277.

[8] Kratkiewicz, K. R. L. (2005) Using a diagnostic corpus of c programs to evaluate buffer overflow detection by static analysis tools. *BUGS'05 Workshop on the Evaluation of Software Defect Detection Tools.*

[9] Pincus, J. and Baker, B. (2004) Beyond stack smashing: recent advances in exploiting buffer overruns. *IEEE Security and Privacy Magazine*, **2**, 20–27.

[10] Anderson, P. (1972). Computer security technology planning study. Technical Report ESD-TR-73-51, U.S. Air Force Electronic Systems Division, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC).

[11] Erlingsson, U. (2004) The Inline Reference Monitors Approach to Security Policy Enforcement. PhD thesis Cornell University.

[12] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. (2009) Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security*, **13**, 1–40.

[13] Aho, A. V., Sethi, R., and Ullman, J. D. (2006) *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[14] (1998). Glossary of telecommunication terms. Federal Standard 1037C, soft-cover, Government Institutes.

[15] Puschner, P. P. and Schedl, A. V. (1997) Computing Maximum Task Execution Times -A Graph-Based Approach. *Real-Time Systems*, **13**.

[16] Common vulnerability enumeration.

[17] Common vulnerability enumeration.

[18] MCCAMANT, S. and MORRISETT, G. (2005). Efficient, verifiable binary sandboxing for a cisc architecture. Tech. Rep. MIT-LCS-TR-988, MIT Laboratory for Computer Science.

[19] HEINTZE, N. and TARDIEU, O. (2001). Ultra-fast aliasing analysis using cla: A million lines of c code in a second.

[20] D. Avots, V. B. L. M. S. L., M. Dalton (2005) Improving software security with a c pointer analysis. *Proceedings of the 27th international conference on Software engineering*.

[21] Zhang, T.-P. (2006) Radar: compiler and architecture supported intrusion prevention, detection, analysis and recovery. PhD thesis Georgia Institute of Technology.

[22] Reis, G. A., Chang, J., Vachharajani, N., Rangan, R., and August, D. I. (2005) Swift: Software implemented fault tolerance. *Proceedings of the International Symposium on Code Generation and Optimization*, Washington, DC, USA CGO '05, pp. 243–254. IEEE Computer Society.

[23] Standard performance evaluation corporation.

[24] Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., and Zhang, Q. (1998) StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. *In Proceedings of the 7th USENIX Security Symposium*, pp. 63–78.

[25] Chiueh, T.-C. and Hsu, F.-H. (2001) RAD: A Compile-Time Solution to Buffer Overflow Attacks. *21st International Conference on Distributed Computing Systems*, pp. 409–417.

[26] Necula, G. C., Condit, J., Harren, M., McPeak, S., and Weimer, W. (2005) CCured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, **27**, 477–526.

[27] JIM, M. G. G. D. H. M. C.-E. J., T. and WANG, Y. (2002). Cyclone: A safe dialect of c. June.

[28] suk Lhee, K. and Chapin, S. J. (2002) Buffer overflow and format string overflow vulnerabilities. *Software: Practice and Experience*, **33**, 423–460.