Control Flow Integrity via Data Flow Integrity Specification and Dynamic Enforcement

Abstract

The main aim of many current software security attacks is arbitrary controlling program behavior at runtime. These attacks often subvert program runtime data flow to subvert its runtime control flow. To prevent such a broad class of attacks that are among the most critical and common attacks, we define a policy that we call Control Flow Integrity via Data Flow Integrity (CFI via DFI). Moreover, based on inline monitoring which is a practical security policy enforcement mechanism, we propose an enforcement approach to dynamically enforce this policy. Therefore, we present some static analysis schemes to instrument programs effectively; they satisfy the policy in a predictive way while any imminent violation is detected, and due to exploiting a recovery technique, program execution continues as if no violation was going to happen. The approach does not have any false positive and is automatically applicable to existing programs. While the most related work has several categories of false negative, our approach has just the false negatives inherited from the basic analysis techniques and common in all of the related work. Our evaluation on SPEC CPU 2000 and 2006 benchmarks and our reasoning show that the policy satisfaction prevents attacks in both of real and synthetic exploitable code, and in the RIPE benchmark. Its overhead is quite reasonable and in case of considering situations of real attacks, it is negligible.

Index Terms

Control Flow Analysis, Data Flow Analysis, Integrity, Inline Monitoring, Instrumentation, Security Policy, Dynamic Enforcement.

I. Introduction

Program runtime *Control Flow* (CF) in the literature refers to the transition of control among a sequence of instructions at runtime that leads to their execution. In each program execution, such a sequence through which the program runtime CF passes is called the *runtime path*. For each program, there is a directed graph known as Control Flow Graph (CFG) [1] containing all its possible sequences of instructions, called *CFG paths*, through which the program runtime CF may pass. CFG of programs can be extracted statically.

Programs are often subject to attacks with the aim of subverting the program CF. To prevent some of these attacks, Control Flow Integrity (CFI) as a security policy is introduced in [2] which dictates software execution must follow a path of its CFG. According to the existing attacks, although following a path that belongs to the program CFG is a necessary condition to satisfy the integrity of control flow, it is not sufficient because based on the semantics of instructions and current state of the program at each of its execution, including the program inputs, there is only one particular path of the program CFG that must be followed in case of no attack. We call such path of the CFG for each program execution the *true execution path*. To subvert the runtime CF from the true execution path to another one, probably with more or special privileges, many attacks subvert the program data flow integrity (and not necessarily data integrity which is different [14]) as a preliminary step [4], [6], [7]. For example, attackers can exploit buffer overflow to overwrite critical memory locations such as return addresses or some program variables such as decision-making data to subvert runtime CF to attackers' desired path. For a better understanding of these attacks, we define the part of program data that is used to select the path of CFG at runtime which is not necessarily an address, *control-data*, and we divide it into two subcategories: *address-control-data* and *non-address-control-data*, and we call the rest of program data *non-control-data* which are explained in the next subsection.

A. Control-Data

Previous work assume that there are two categories of data: control-data which is referred to the data that is loaded into the processor program counter or any data used for calculating such data, e.g. return addresses and function pointers [3], and other data which are called non-control-data [6]. Moreover, attacks that overwrite control-data are called control-data-attacks, and some others exploiting similar vulnerabilities to overwrite security critical data without subverting the intended CF in the program are non-control-data attacks [6]. According to these definitions, some critical data such as user identification data or decision-making data are neither control-data nor non-control-data because they are not loaded into the processor program counter while attacks that overwrite them can also subvert the intended program CF (as is shown in an example in II-A).

We believe that this problem arises because the categorization of control-data and non-control-data does not completely correspond to the CF transition in programs. CF transition in a program is based on its CFG [1] which is widely used in the field of program analysis [1], [2]. Briefly, during analyzing the whole program, CFG refers to a graph whose vertices are blocks of one or more instructions, and its edges depict the control transition between two blocks of instructions. As noted before, if a data is used to choose a path by the program CF at runtime, we call it *control-data*. By this definition, control-data includes data that are directly loaded into the program counter, e.g. a jump target or a function pointer, which we call *address-control-data*. There is other data, e.g. user identity data or decision-making data, that although are used to choose the next

block of instructions, are not an address to directly be loaded into the program counter which we call *non-address-control-data*. Moreover, if a data is not used to choose a path by the program CF at runtime, we call it *non-control-data*. Similarly, we call attacks that overwrite any kind of control-data *control-data-attacks*.

In control-data-attacks, the program CF can be subverted simpler and easier to detect in case of address-control-data attacks than non-address-control-data attacks [12]. It is because by changing an address-control-data an attack can be done without knowing semantic details of vulnerable programs. In addition, detection of such attacks is easier because their main aim is usually running a code which is never intended to be run and thus they can be detected easier. In contrast, in non-address-control-data attacks, arbitrary data overwriting usually results in running a sequence of instructions of the program that although could be run in some cases, is not intended to be executed at a particular execution. This situation makes the attack detection more complicated. Let us call these two kinds of violations, *Outbound* and *Inbound* CFI via DFI violations, respectively. However, accomplishing an inbound violation by attacks is not a limitation for powerful attackers because the semantics of widely used applications are available, and their understanding is simply possible through analysis tools. Since examples of address-control-data attacks can be seen easily, we show an example of inbound CFI via DFI violations through non-address-control-data attacks.

Most of useful techniques such as [2], [10] just pay attention to attacks against address-control-data, and they cannot defend non-address-control-data attacks [4], [7]. However, attacks against non-address-control-data, which are previously considered as non-control-data, are realistic and critical [6]. A few live exploits with original well-known vulnerabilities which are non-address-control-data attacks are included but not limited to widely used `SSH` which overwrites an authenticated variable [4], the most widely used `FTP` server `WU-FTPD` [11], exploiting user identification data in `Portmap`, `Sendmail` [12], `Telnetd` [12], and `Cfengine` [13].

There are some techniques with the aim of detection or somehow prevention from attacks against the integrity of data flow which consider all program data including non-control-data, e.g. [4], [7], [14]. The main difference between them and our presented method is that our focus is on control-data flow integrity to provide the integrity of CF. Moreover, approaches such as [4], [7] have false negatives that our approach does not, and impose higher overhead especially in [14].

In this paper, our main aim is preventing attacks that subvert the integrity of CF at runtime to follow a sequence of instructions that do not correspond to the CFG true execution path which we define based on the integrity of control-data flow. With this aim, we specify and enforce a policy that we call CFI via DFI. Generally, there might be more constraints to explain the true execution path. However, according to security advisories and real critical attacks, we determine one of the most important constraints on runtime paths which is DFI [14], [6], [3], and impose it through the enforcement of our policy to prevent such attacks. Moreover, almost all well-known exploitable vulnerabilities in these attacks making them alive threats, have historically accounted for a significant percentage of the software vulnerabilities published periodically in NIST, OWASP top ten, CERT advisories, CVE lists and etc., while they have medium or high severity rating in years.

To specify CFI via DFI, we analyze CFG paths, and determine instructions that according to control-data can choose the next instruction in the CF and we call them *Control Flow Selector Instructions* ($cfsi$). Briefly, CFI via DFI dictates that each $cfsi$ in each program execution, based on DFI satisfaction for the used control-data in that instruction, should choose the true ongoing sub-sequence of instructions. Then, if all such sub-sequences are truly chosen by $cfsi$s in a program execution, then according to the policy, CF passes through the true runtime path at that execution. To enforce CFI via DFI at runtime, we rely on Inline Reference Monitoring (IRM) that merges the code of the enforcement mechanism, i.e. security code, into the target code.

Let us call a precise dynamic slice of executed instructions at runtime, that could directly cause a CFI via DFI violation in a $cfsi$, the *$cfsi$ violation sub-path*. Although finding vulnerabilities is not an aim of this paper, the approach is proposed such that, any imminent violation and thus a vulnerable code is detected, and it is possible to provide a precise violation sub-path if the runtime path is logged. Such detection enables web software engineers to prioritize security auditing efforts [25]. In addition, dynamic slice significantly reduces the analysis efforts to find an existing or a new vulnerability. However, our approach and vulnerability-based techniques can be complementary approaches in satisfaction of our desired policy.

To briefly review the approach, we instrument the program such that legal values of used control-data for each $cfsi$ is maintained and used to detect violation from following the true path by path selections in $cfsi$s. The last place that such violation can be detected before happening for each $cfsi$, i.e. its detection point, is right before the $cfsi$ itself. Moreover, for each $cfsi$ at runtime, a violation sub-path is started after the first definition of one of its used control-data reaching to that $cfsi$, to its detection point. In addition, the longer the distance between that definition and the detection point is, the longer the violation sub-path becomes.Therefore, to make earlier detection and shorter violation sub-path for each $cfsi$ possible, we need to predict each $cfsi$. Moreover, to make them earliest and shortest, we need to find the first place that each $cfsi$ is predictable, and then the approach can have several detection points from the prediction point to the point before the $cfsi$. However, due to the aim of this paper which is prevention of the policy violation, we simplified the approach by choosing the last place to detect the violation before happening. Hence, CFI via DFI dictates that each $cfsi$ in each program execution should choose the ongoing sub-sequence of instructions corresponding to its predicted one. In such cases, the execution path is considered true. To enforce the policy in more details, for every $cfsi$ we find the first place in every runtime path such that by guaranteeing DFI satisfaction for required control-data, the path selection by that $cfsi$ is predictable. Then, we instrument the program to

predict path selection for every $cfsi$ at every runtime path. In addition, immediately before every actual path selection by each Control Flow Selector (CFS) instruction, inserted code detects whether the path selection is going to be the same as the predicted one. If not, it guides the runtime control flow to the true path while any violation from CFI via DFI is prevented. Moreover, to guarantee DFI satisfaction for required control-data, the inserted code through instrumentation traces control-data manipulations along every path and makes sure that they are legal. Otherwise, control-data will be recovered to avoid the policy violation. Therefore, the remedial action is data recovery which makes the runtime CF follow the true CFG path. Due to having guides for the runtime control flow to the true path, and data recovery, the program execution after detection of an ongoing violation continues while the violation is prevented, as if no violation was going to happen.

The CFI via DFI enforcement approach is presented in two main phases. In the first phase we propose some data flow analysis algorithms and use them to find the effective insertion points to inline security checks such that any imminent violation of CFI via DFI is detected dynamically as soon as possible before happening. The algorithms are as follows:

- *Path Reaching Definition Algorithm*: It is an extension of reaching definition algorithm in [1] that in addition to determining all definitions reaching a block, it returns all sub-paths from which each definition reaches the block.
- *Control Flow Selector Prediction Point Algorithm*: It finds prediction points for each $cfsi$ in each runtime path.
- *Control Flow Selector Variable Algorithm*: It finds all variables that directly or indirectly may affect the path selection at runtime.

In the second phase, we propose instrumentation steps to inject security code in insertion points. Moreover, we support static and dynamic linking of compiled modules, when an application is divided into modules that might be a library. Each module includes its own code and data that can be statically analyzed and instrumented separately. Moreover, to link modules together, for each module some auxiliary information (e.g. data that can flow out of the module) are extracted. Then, these modules could be linked together while their auxiliary information are combined and the instrumentation still guarantees the truth of data flows. This approach is also used in Modular Control-Flow Integrity (MCFI) [28], and our approach uses the same technique to support linking between modules. In addition, considering multi-thread settings, it is possible for different threads to access to the same auxiliary information of a module. To prevent possible interferences in such cases, we also provide a critical section for shared information such as the auxiliary information.

We measured runtime and memory overhead of the CFI via DFI enforcement for CPU intensive SPEC 2006, and programs of the SPEC 2000 benchmark [15] that are common among the most related work to facilitate comparison with them in Windows XP SP2 which is a close platform to those techniques. Our evaluation and reasoning show that the policy satisfaction prevents attacks in both of real and synthetic exploitable code, and in the RIPE benchmark [31]. Even by supporting multi-threading and linking in our approach which are not supported by the compared related work, the imposed overhead is between the lowest and the highest overhead of the related techniques. Moreover, by analyzing control-data and its categorization, and considering real target data in CFI via DFI attacks, we propose an optimization for the enforcement approach which results in a significant overhead reduction to a negligible amount.

To the best of our knowledge, there is no recovery technique in previous work. Moreover, assumptions, specification and guarantees of the policy are established explicitly. In addition, while we explain that the most related work has several categories of false negative, our approach has just the false negatives based on the used analysis technique which is common with, and less than one of those categories.

The rest of the paper is structured as follows. Section II describes preliminaries and defines the CFI via DFI policy. The enforcement approach is presented in Section III. Section VII discusses the most related work. Section VIII presents the evaluation of our enforcement approach. Finally, Section IX concludes the paper and briefly describes future work.

II. CFI via DFI Specification

To specify the CFI via DFI policy in this section, we redefine control-data in more details in II-A, and explain a CFI via DFI violation in the boundary of CFG which we call *inbound CFG violation*, with an example. Then, we describe CFI from our point of view based on CFG, and review DFI based on which we define a simplified DFI satisfaction condition. Finally, by providing required definitions and information about $cfsi$s, data dependencies, control-data and path prediction points, we specify the CFI via DFI policy.

A. Example of an Inbound CFI via DFI Violation

The source code representation of a simplified code fragment in Fig. 1 shows a synthetic example partially extracted from real exploitable codes such as `WU-FTPD` and `SSH`.

In this code fragment, there is a variable $auth$ which, probably according to the user account information, is assigned a value of 0 or 1. Then, the assigned value to this variable is used to grant or deny the access of system resources to the user, through instruction 8. Therefore, this variable is a non-address-control data based on which runtime CF chooses its ongoing sequence of program instructions. If exp is false, the value of $auth$ is assigned by instruction 1, and otherwise by instruction 4. In the case of no attack, if $auth$ is set to 0 or 1 then the runtime CF in instruction 8 takes the sequence of instructions related to access denial or access grant, respectively which is predictable after $auth$ assignment. The real path selection in instruction 8 should correspond its prediction.

```
1: auth=0
2:
3: if (exp)
4:     auth= ...
5:
6: Strfunc(buffer)
7:
8: if (auth==1)
9:    Access-grant(unam)
10: else
11:    Access-deny(unam)
12:
```

a)

```
      _auth = ASSIGN 0                          #1
$L6:  t274 =CMP(NE) _exp, 0                     #2
      CBRANCH(NE) t274, $L8, $L7                #3
$L8:  _auth = ASSIGN ...                        #4
$L7:  tv275 = CALL &_Strfunc, &_buffer          #5
$L9:  t276 =CMP(NE) _auth, 0                    #6
      CBRANCH(NE) t276, $L11, $L10              #7
$L11: tv277 = CALL &_access-grant &_unam        #8
      GOTO $L12                                 #9
$L10: tv278 = CALL &_access-deny &_unam         #10
$L12:
```

b)

Fig. 1. a) Example of a vulnerable code fragment to non-address-control-data attack with CFI via DFI violation, and b) its HIR form.

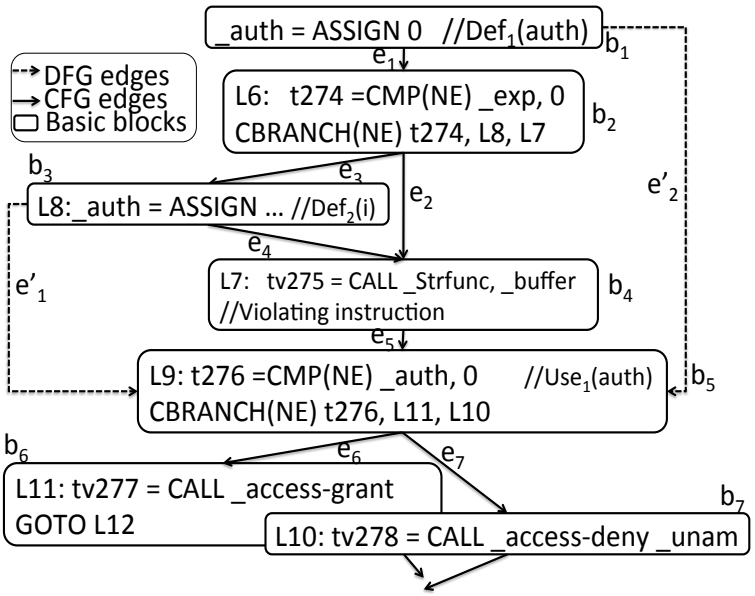


Fig. 2. CFG and DFG of the example in Fig. 1.

The code fragment is vulnerable to buffer overflow because of `Strfunc` which allows extra data to overwrite other memory contents, e.g. it can make an arbitrary change in $auth$, and thus can cause a runtime path that according to legal variable manipulation is not true. For an instance, if $auth$ is legally assigned 0, then the path selection in line 8 can be predicted to the sub-path related to the access denial, while by exploiting the vulnerability, it can be changed to 1 that can cause following the sub-path related to the granting access to an unauthorized user by an attack.

B. Control Flow Integrity

To statically extract a program CFG before its execution, its instructions are divided into basic blocks [1]. The CFG associated with the program is a graph $G = (B, E)$ such that B is the set of basic blocks including a starting and terminating vertices, i.e. *s* and *t*, each of which is identified by a unique number, and E is the set of edges such as $e_m = (b_i, b_j)$ showing a possible transition of control between two basic blocks. A program *path* is a sequence of vertices and edges from s to t. Moreover, P is the set of all the program CFG paths each of which is denoted by p_i while i uniquely identifies each path.

According to CFI [2], a runtime path (RP) of a program is a *True Runtime Path* if $RP \in P$ which means that runtime path RP should belong to the set of all possible runtime paths of the program CFG. If a runtime path does not satisfy this condition, i.e. the runtime CF is subverted to a sequence of instructions that does not correspond to any CFG path, then an outbound CFG violation occurs. As noted before, even if this condition is satisfied, there are non-trivial security attacks that subvert the runtime CF from a path of CFG to another, i.e. attacks that can cause inbound CFG violations. To prevent these attacks, it is required to explain true runtime path of CFG by some conditions, and propose techniques to ensure satisfaction of those conditions at runtime. Our main aim in this paper is to prevent a broad class of these attacks by focusing on a necessary condition of a true path of the program CFG related to the integrity of program data flow.

C. Data Flow Integrity

Every instruction that based on its semantics assigns a value to a variable v is called a definition instruction [1]. According to [14], such instructions are marked as $Def(v)$, and every instruction that uses v is marked as $Use(v)$ with a unique number. For every instruction or block x, $x.ML$ denotes the set of all its marks.

DFG of a program with a CFG $G = (B, E)$ is a graph $G' = (B, E')$ such that E' is a set of edges, e.g. $e'_n = (b_i, b_j)$ denoting an edge from b_i to b_j if and only if for a variable v in the program, $Def(v) \in b_i.ML$ and $Use(v) \in b_j.ML$, and there exists at least one path, including a sub-path from b_i to b_j, in which there is not any legal definitions for v in between. In such a case, it is said that there is a data flow from b_i to b_j denoted by $df : Def(v) \rightarrow Use(v) \in DFG$. A runtime data flow, rdf, in a runtime path from b_i to b_j occurs if an instruction in b_i assigns a value to a variable v that is used in an instruction in b_j without any change in its value. For example, the CFG and DFG of the vulnerable code fragment in Fig. 1(a) are shown in Fig. 2 while its high-level intermediate representation (HIR) is shown in Fig. 1(b).

A runtime data flow, rdf, in a runtime path $RP \in P$ is a *true data flow* if two conditions are satisfied: first, $rdf \in DFG$ which means that there exists a non-empty set of CFG paths respectively passing through b_i and b_j, while an instruction in b_i and another one in b_j legally defines and uses the variable according to their semantics, and second, RP is one of those paths. In such cases, it is said that rdf is an expected data flow in those paths. According to this definition, although there are two $df : (Def_1)(v) \rightarrow (Use_1)(v)$ and $df : (Def_2)(v) \rightarrow (Use_1)(v)$ in the program DFG in Fig. 2, if exp is always false at runtime, the second data flow could never be a true runtime data flow. The DFI is a policy that dictates every software runtime data flow should be a true data flow. According to this definition of DFI, we extract a property of TDFs, which we call the *DFI Satisfaction Condition*, explained as follows.

Definition 1 (DFI Satisfaction Condition)**.** *In every true data flow $rdf : Def(v) \rightarrow Use(v) \in RP$, the DFI satisfaction condition states that the value of v after the instruction marked as $Def(v)$ and before the instruction marked as $Use(v)$ at runtime must be the same.*

D. CFI via DFI

To define the CFI via DFI policy, we present some definitions.

Definition 2 (Control Flow Selectors)**.** *We call each $b \in CFG$ a Control Flow Selector Block, denoted by $cfsb$, if there are more than one output edge from b, and b includes an instruction that, according to an expression using a set of variables, e.g. $\{v_1, ..., v_i\}$, chooses the next edge from its output edges. Such an instruction, expression, and variable are respectively called Control Flow Selector Instruction ($cfsi$), Control Flow Selector Expression ($cfse$), and direct Control Flow Selector Variable ($cfsv$).*

According to the above definition, instructions such as `if` and `while` are $cfsi$s. Let us call the set of all such $cfsv$s and all $cfsb$s of a program, *direct $CFSV$* and *$CFSB$* sets, respectively. Direct $CFSV$ consists of all used variables in all $cfsb$s because these variables directly affect path selection at runtime. Furthermore, there can be a set of variables that may indirectly affect variables in direct $CFSV$, and thus, affect the path selection at runtime which we call *indirect $CFSV$*. All variables in direct $CFSV$ and indirect $CFSV$ are control-data. To find indirect $CFSV$, we need to define *Data Dependency* and *Control Dependency* as follows.

Definition 3 (Data Dependency)**.** *If an instruction, I, that defines a variable v, i.e. $Def(v) \in ML(I)$, uses another variable v' to define v, i.e. $Use(v') \in ML(I)$, then it is said that v is data-dependent on v' or there is a data dependency from v' to v.*

If v is data-dependent on v' and the instruction I that defines v belongs to b_2, and v' is defined in b_1, we also say that b_2 is data-dependent on b_1. For example, if in the vulnerable code, instruction 1 is $auth = zero$ when $zero$ is another variable, then $auth$ is data-dependent on $zero$, and b_1 would be data-dependent on the block containing the $zero = 0$ preceding b_1. Moreover, this data dependency between blocks is similar to true dependence defined in [1] which implies that there is a true dependence from b_1 to b_2 if there is a write to a location in b_1, e.g. a definition for v', that is followed by a read of the same location in b_2, e.g, a use of v'.

Definition 4 (Control Dependency)**.** *If there is an instruction I that uses a variable v' to determine whether an instruction that defines another variable v is executed, then it is said that v is control-dependent on v' or there is a control dependency from v' to v.*

Instructions that can determine the execution or skip of another instruction have more than one output edge. Again, if v' and v are two variables that $Use(v') \in ML(b_1)$ and $Def(v) \in ML(b_2)$, and v is control-dependent on v', then we say b_2 is control-dependent on b_1. This dependency between blocks corresponds to the defined control dependency in [16] that states a block is control-dependent on a branch if one edge from the branch definitely leads to the execution of instructions in that block, while another edge can lead to skipping that block. Therefore, instructions in that block cannot be executed until its immediate control dependency branch is executed. For example in Fig. 2, if exp is an expression that uses a variable such as m, then it is said that variable $auth$ is control-dependent on variable m, and b_3 is control-dependent on b_2. Indirect $CFSV$ is the set of all variables v' such that there is a data dependency or control dependency from v' to any $v \in$ indirect and direct $CFSV$, if collecting this set starts from an empty set for indirect $CFSV$.

Let us assume that in each runtime path p_i, every variable should be defined at least once before its use. Therefore, for each $cfsb$ in each p_i, there is a sequence of instructions, not necessarily consecutive instructions in p_i, preceding the $cfsb$; we call this sequence $cfsb.DefSeq \in p_i$ where:

- for each v that $Use(v) \in cfsb.ML$, $cfsb.DefSeq$ includes a definition instruction marked as $Def(v)$ such that there is a data flow $Def(v) \rightarrow Use(v)$ in DFG and the definition is the last legal definition of v before $cfsb$ in p_i.

So, each runtime path RT consists of arbitrary numbers of $(cfsb.DefSeq,\ cfsb)$ pairs.

Definition 5 (Prediction Point (PP))**.** *For each $cfsb \in CFSB$ of a program with the pair of $(cfsb.DefSeq,\ cfsb)$ in the runtime path RP, the program point (denoted by $cfsb.PredPoint$) after the last definition instruction of the $cfsb.DefSeq$ in the RP is the first place that the result of the $cfsi$ in the $cfsb$ is predictable in this path if by using all used variables in the $cfsb$ in that point, DFI and thus its satisfaction condition is satisfied for:*

- *all data flows from $Def(v) \in cfsb.DefSeq$ to the $cfsb$ when $v \in$ direct $CFSV$, $Use(v) \in cfsb.ML$, and*
- *all data flows in the RP preceding the $cfsb.DefSeq$ related to each variable $v' \in$ indirect $CFSV$ that used variables in $cfsb$ are dependent on.*

We call this point of the program, the $cfsb$ prediction point in the RP.

If an enforcement approach guarantees DFI satisfaction condition, it is obvious that the $cfsb$ PP in each runtime path RP is the first place in which used variables in the $cfsb$ are assigned their final value, and thus, the $cfsb$ path selection can be predicted by pre-determination. In such a case the CFI via DFI policy is defined as follows.

Definition 6 (CFI via DFI Policy)**.** *CFI via DFI Policy is a security policy that dictates in each $cfsb \in CFSB$ of a program in each runtime path $RP \in P$, the $cfsb$ path selection corresponds its predicted path which is determined in its PP in the RP.*

Definition 7 (CFI via DFI true path)**.** *From the CFI via DFI policy point of view, a runtime path RP is a true runtime path if and only if it satisfies CFI via DFI Policy.*

III. CFI via DFI Enforcement

To dynamically enforce a policy through inline monitoring, it is required to provide a protected part to save related program behavior [8]. We provide such a protected part by checking if the address of the location being defined is in the memory of the protected part. If so, for such instructions an exception is raised. Moreover, we use an extra segment register such as the FS register with a non-zero base address. Then, we modified the loader to fetch the FS register for the protected memory which will be located after a 4KB guard page following the instrumented program in the virtual address space and thus, the starting address of the protected part is FS+4KB. Therefore, to find the address of the protected part for a variable, its offset must be added to the address of the protected part. Moreover, it is required that the target address of definitions that are obtained by the sum of the offset of the used segments and the size of the defined data, to be less than the size of the located guard page. Therefore, to provide the protected part it is sufficient to check if the bitwise `AND` of FS with the address of definitions is non-zero. Moreover, we assume that the program code is Non-Writable Code (NWC) by using read-only protection for code pages, and No Other programs can Change (NOC) the program data memory.

The CFI via DFI enforcement approach is presented in two main static phases including *Control and Data Flow Analysis Schemes* and *CFI via DFI Instrumentation*. In the first phase, insertion points are found based on flow analysis, and in the second phase instrumentation steps are explained.

A. Control and Data Flow Analysis Schemes (First Phase)

In the reaching definition algorithm, for each $Use(v)$, a set of all instructions marked as $Def(v)$ that may have a true runtime data flow to that use can be collected. To find all prediction points for a $cfsb$ in each possible path containing the $cfsb$, it is needed to find $cfsb.DefSeq$s. To do so, first we present the *Path Reaching Definition Algorithm* based on the reaching definition which in addition to determining all definitions that reach a block, returns all sub-paths from which each definition reaches it. Then, the *CFS PP Algorithm* is presented to find all $DefSeq$s of a $cfsb$ belonging to different paths containing the $cfsb$. Using this algorithm, the PP for each $DefSeq$ sub-path of the $cfsb$ is determined.

1) Path Reaching Definition (PRD) Algorithm: The PRD algorithm is presented in Alg. 1 while the sets $gen_b, kill_b, IN[b]$ and $OUT[b]$ have the same definitions as the reaching definition algorithm summarized in the next subsection.

IV. Review of PRD Algorithm Based on Reaching Definition

PRD algorithm is presented in Alg. 1. It is said that a definition of a variable x which is made at instruction d is killed along a path if there is another definition of x following the former definition. Furthermore, it is said that a definition of a variable x which is made at instruction d reaches a block b if there is a path from the point immediately following instruction d to block b such that this definition is not killed along that path. In this algorithm, all definitions in the program are listed such as Def_1, Def_2, ..., Def_n. Moreover, for each block b there are four sets $gen_b, kill_b, IN[b]$ and $OUT[b]$ including some of those definitions. the algorithm uses gen_b and $kill_b$ sets for each block and generates $IN[b]$ and $OUT[b]$ in an iterative

Algorithm 1: PRD Algorithm.

Input:
gen_b: generated definitions in block b,
$kill_b$: killed definitions in block b
Output:
$IN[b]$: definitions reaching the entry of block b,
$OUT[b]$: definitions reaching the exit of block b
OUT[ENTRY] = 0
for *(each basic block b other than ENTRY)* **do**
    OUT[b] = 0
end
while *(changes in any OUT bit vector occur)* **do**
    for *(each basic block b other than ENTRY)* **do**
        $IN[b] = \bigcup_{join\,(P\,a\,predecessor\,of\,b)}(OUT[P])$ // (1) $OUT[b] = gen_b \bigcup_{pass}(IN[b] - kill_b)$ // (2)
    end
end

approach similar to the reaching definition algorithm. Gen_b for each block b represents definitions generated by instructions in b. Therefore, if there are just two definitions Def_1 and Def_2 in a program, then this sets for each block is modeled as a couple, e.g. $gen_b = (0, 1)$ which means Def_2 is generated in b. Also, $kill_b$ includes the set of definitions that might be killed because of definitions in this block. Therefore, $kill_b$ consists the set of all definitions of each variable x in the program that is also defined in b. In $IN[b]$, each bit is 1 if its corresponding definition reaches the entry of b, and is 1 in $OUT[b]$ if it passes b and reaches its exit without being killed.

The PRD algorithm begins with estimating $OUT[b] = 0$ for all b and iterates until all OUTs and hence INs converge through $while - loop$. Therefore, each iteration starts after finding that there is, at least, one OUT which has been changed for a block. The inner loop with for structure applies the data-flow equations to each block. Intuitively, simulating all possible executions of the program, this algorithm propagates definitions as far as they will go without being killed. Data-flow equations composed of equation (1) and (2) in Alg.. 1. In the first equation and for the first block, which can be any block of the CFG, definitions that reach the exit of all its predecessor blocks constitute the IN set of the block. Then, in the second equation, all definitions that can pass through the block without being killed are joined to the set of generated definitions in the block to constitute the set of definitions that reach the exit of the block in its OUT set. These two data-flow equations are computed for all blocks. While there is, at least, one block that its OUT is changed, then another iteration starts. In general, a bit vector, e.g. OUT of a block, is changed if at least one of its bits regardless to its subscript is changed. Since sets in this algorithm are shown as bit vectors, the union of sets is computed by taking the logical OR($\vee$) of their corresponding bit vectors, and the difference of two sets is computed by taking the logical AND between the bit vector of the first set and the complement of the second one.

A. Equations in PRD Algorithm

There are operators such as $\cup_{join}$, subtraction and $\cup_{pass}$ in data-flow equations of the PRD algorithm that are applied on bits of two bit vectors. Possible inputs and outputs of these operators are shown in Table I. In this table, each two bits are denoted by a and c, and their subscripts are denoted by p_1 and p_2. Computation over subscripts is explained using if-statements.

To calculate $Out[b_n]$ in equation (2), $\cup_{pass}$ represents the effects of definitions in a block on definitions that pass through it by using two subscript manipulation functions $extend$ and $concat$. These functions take a set of sub-paths p and a block b_n as their inputs and return another set of sub-paths p' as their output as the following. Table I shows how $\cup_{pass}$ works by using these functions.

- function $extend$ returns the extension of p by adding b_n as a new sub-path, e.g. in $p' = extend(p, b_n)$ $p' = p, b_n$, and
- function $concat$ returns the concatenation of each sub-path $b_{i...j} \in p$ and b_n, e.g. in $p' = concat(b_{i...j}, b_n)$ p' contains $b_{i...jn}$.

According to possible cases of bits as operands of $\cup_{pass}$ in Table I, if a definition is generated at block b_n, then the existing path of reaching definitions should be extended by the new sub-path. Also, if a definition already reaches b_n, its corresponding bit in $IN[b] - kill_b$ in Table I is shown by c, is 1, then all existing sub-paths should be connected with b_n to represent passing the definition through this block too.

1) (PRD) Algorithm: Each set is modeled as a bit vector with the form of $(bit_{1\,ssp_1}...bit_{n\,ssp_n})$ when n is the number of definitions and each ssp_i for each $1 \leq i \leq n$ is a set of sub-paths as a subscript. Each bit of these bit vectors corresponds a definition and is 1 if and only if that definition is included in the set, and is 0 otherwise.

TABLE I
BIT OPERATIONS IN PRD ALG.

Formula	Example
$IN[b] = OUT(p) \cup_{join} OUT(p') = a_{p_1} \cup_{join} c_{p_2} = (a \vee c)_{p_1 \cup p_2}$	$1_{b_i} \cup_{join} 1_{b_j} = 1_{b_i,b_j}$ $0 \cup_{join} 1_{b_j} = 1_{b_j}$ $0 \cup_{join} 0 = 0$
$IN[b] - kill_b = a_{p_1} - c_{p_2} = (a - c)_p$ subscripts computation: $if \quad (a - c) = 0 \ \ then \quad p = empty$ $else \quad p = p_1$	$(0, 1_{b_j,b_k}) - (0, 1)$: $1_{b_j,b_k} - 1 = 0$ $1_{b_j,b_k} - 0 = 1_{b_j,b_k}$
$OUT[b_n] = gen_{b_n} \cup_{pass} (IN[b_n] - kill_{b_n})$ $= a \cup_{pass} c_{p2} = (a \vee c)_p$ subscripts computation: $if \quad c = 1 \quad then \quad p = concat(p_2, b_n)$ $if \quad a = 1 \quad then \quad p = extend(p_2, b_n)$	$(0, 1) \cup_{pass} (0, 1_{b_j,b_k})$: $0 \cup_{pass} 0 = 0$ $0 \cup_{pass} 1_{b_j,b_k} = 1_{b_{jn},b_{kn}}$ $1 \cup_{pass} 0 = 1_{b_n}$ $1 \cup_{pass} 1_{b_j,b_k} = 1_{b_{jn},b_{kn},b_n}$

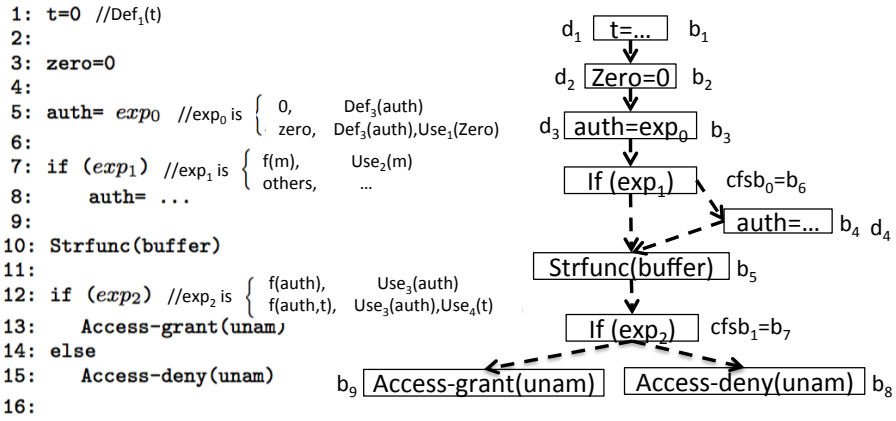


Fig. 3. Extended example of a vulnerable code in Fig. 1, its CFG and DFG.

To cover and explain different cases in our analysis algorithms, we extend the example in Fig. 1 by assuming two possible cases for the expressions shown in Fig. 3. Let us consider the simplified CFG of this example which is sketched using the source code instructions to avoid unnecessary details. There are four definitions Def_1 defining t, Def_2 defining $zero$, Def_3 and Def_4 defining $auth$. Table II shows bit vectors of gen_b and $kill_b$ for blocks in this example. $Gen_{b_3} = (0010)$ in this table means that definitions Def_3 for variable $auth$ is generated at block b_3. Also, $kill_{b_3}$ is 0001 which shows if Def_4 that also defines $auth$ reaches the entrance of block b_3, it will be killed in this block. As an example of a bit subscript in $IN[b]$, without details about how it is computed, consider the bit and its subscript corresponding to the definition Def_1 in $IN[cfsb_1]^1$ which is $1_{b_{1236},b_{12364}}$. This implies that the first definition reaches the beginning of $cfsb_1$ through the sub-path $b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b_6$ and also through another sub-path $b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b_6 \rightarrow b_4$. The PRD algorithm shows how these sub-paths are computed.

Data-flow equations in the PRD algorithm have operators such as $\cup_{join}$, subtraction and $\cup_{pass}$ that are applied on bits of two bit vectors. Possible inputs and outputs of these operators, and their details are explained in Section IV-A. $\cup_{join}$ in equation (1) of the algorithm is used to compute $IN[b]$ for each b if it has more than one predecessor block in the CFG. Since $\cup_{join}$ should represent all sub-paths from which a definition can reach a block, it returns the union of corresponding bits and their subscripts. Furthermore, the data flow equation $IN[b] - kill_b$, in equation (2), is used to show effects of killing definitions

TABLE II
COMPUTATION OF IN AND OUT FOR THE CODE IN FIG. 3.

Blc.	gen	kill	$IN[b]^1$	$OUT[b]^1$
b_1	10 00	00 01	00 00	1_{b_1} 0 00
b_2	01 00	00 00	1_{b_1} 0 00	$1_{b_{12}}$ 1_{b_2} 00
b_3	00 10	00 01	$1_{b_{12}}$ 1_{b_2} 00	$1_{b_{123}}$ $1_{b_{23}}$ 1_{b_3} 0
$cfsb_0 = b_6$	00 00	00 00	$1_{b_{123}}$ $1_{b_{23}}$ 1_{b_3} 0	$1_{b_{1236}}$ $1_{b_{236}}$ $1_{b_{36}}$ 0
b_4	00 01	00 10	$1_{b_{1236}}$ $1_{b_{236}}$ $1_{b_{36}}$ 0	$1_{b_{12364}}$ $1_{b_{2364}}$ 01 b_4
$cfsb_1 = b_7$	00 00	00 00	$1_{b_{1236},b_{12364}}$ $1_{b_{236},b_{2364}}$ $1_{b_{36}}$ 1_{b_4}	$1_{b_{12367},b_{123647}}$ $1_{b_{2367},b_{23647}}$ $1_{b_{367}}$ $1_{b_{47}}$

Algorithm 2: CFS PP Algorithm.

```
Input:
IN[b]: definitions reaching the entry of block b,
Used[b]: variables that are used in block b
Output:
DefSeqL[b]: List of all DefSeq of block b as a cfsb,
PredPoint[b]: List of all Prediction Points of block b as a cfsb
for each block b as a cfsb ∈ CFSB do
    for each used variable uv in Used[b] do
        DefUsed[b][uv] = In[b] ∧ DefBitVec[uv]
    end
end
for each block b as a cfsb ∈ CFSB do
    for each used variable v_1 ∈ Used[b] do
        Ov = Used[b] − v_1
        for each A=1_{path_1} in DefUsed[b][v_1] do
            for all sub-path sp_1 = b_{i...j} ∈ path_1 do
                Mark i in sp_1
                break = true
                for each used variable v_2 ∈ Ov do
                    if (∃ a C=1_{path_2} in DefUsed[b][v_2] with sub-path sp_2 = b_{i'...j'} ∈ path_2) and (sp_2 ∈ sp_1) then
                        Mark i' in sp_1
                    else
                        break = true
                        break
                    end
                end
                if not break then
                    DefSeq = the sequence of marked blocks in sp_1
                    p = latest marked block in sp_1
                    add DefSeq to DefSeqL[b]
                    add p to PredPoint[b]
                end
            end
        end
    end
end
```

by b where a data-flow is passing through b. Therefore, in subtraction, if a definition is killed, the result is 0. Otherwise, the sub-path of a definition that reaches the block and is not killed remains unchanged.

The result of running the PRD algorithm for the CFG in Fig. 3 is represented in Table II. It shows the algorithm is finished after one iteration and path reaching definitions to each $cfsb$ is determined. This algorithm can be used to find all data that reach each module exit point, and all data that can be considered as inputs to the module entry point. Such information is auxiliary information for each module that will be explained later (in the enforcement complementary V-3).

2) Control Flow Selector Prediction Point (CFSPP) Algorithm: The CFSPP algorithm takes $IN[b]$ and all used variables in each $b = cfsb$, i.e. $Used[b]$, and by using all sub-paths that each used variable reaches the $cfsb$ entry, returns the set of all $cfsb.DefSeq$s in $DefSeqL[b]$, and all its prediction points in $PredPoint[b]$.

This algorithm is sketched in Alg. 2. For each variable uv, $DefBitVec[uv]$ is a bit vector that shows which definitions in the program defines this variable. For example, according to the CFG in Fig. 3, $DefBitVec[auth] = (0011)$. The CFSPP algorithm, for each $cfsb$ block b, and for each used variable $uv \in Used[b]$, computes $DefUsed[b][uv]$ that is another bit vector showing all definitions of variable uv that reach b and are used in it. Then, for each $cfsb$, the CFSPP algorithm finds all different $DefSeq$s that precede the $cfsb$ to find all its prediction points. To do so, for each variable $v_i \in Used[b]$, it is checked if each of sub-paths that one of its definitions reaches b, is a super path of a sub-path of a definition of all other variables in $Used[b]$. If it is so, the sequence of definitions in that sub-path are marked by those definitions and all these marked definitions constitute a $DefSeq$ for b which is a $cfsb$. Therefore, such a $DefSeq$ is added to the $DefSeqL[b]$. Moreover, the $cfsb$ PP is right after the last definition of a used variable in the $DefSeq$ which will be added to $PredPoint[b]$.

In the example in Fig. 3, there is one $cfsb$, $cfsb_1$, whose $DefSeq$s is included in the code fragment. According to this figure,

Algorithm 3: Control Flow Selector Variable Algorithm.

Input:
Used[b]: All used variables in block b,
Data-Dep[b]: All blocks with a definition that reach and are used in block b,
Control-Dep[b]: All blocks on which block b's instructions are control-dependent
Output:
DirCFSV: List of all direct CFSV,
IndCFSV: List of all indirect CFSV,
CFSVLst: List of all direct and indirect CFSV
CFSVLst, IndCFSV, DirCFSV=empty
for *all block b in CFSB* **do**
  DirCFSV = DirCFSV ∪ Used[b]
end
for *all block b in CFSB* **do**
  TmpDepBlocks = Data-Dep[b] ∪ Control-Dep[b]
  mark b
end
while *TmpDepBlocks not empty* **do**
  remove b' from TmpDepBlocks
  if *b' is not marked* **then**
    IndCFSV = IndCFSV ∪ Used [b']
    TmpDepBlocks = TmpDepBlocks ∪ Data-Dep[b'] ∪ Control-Dep[b']
    mark b'
  end
end
CFSVLst = IndCFSV ∪ DirCFSV

we assume two possibilities for exp_2 in $cfsb_1$. If exp_2 is $f(auth)$, then $Used[cfsb_1] = auth$. According to this algorithm, $DefSeqL[cfsb_1] = b_3, b_4$. Moreover, if exp_2 is $f(auth, t)$, then $Used[cfsb_1] = auth, t$. In this case, $DefSeqL[cfsb_1] = b_{13}, b_{14}$. In this simple example, in both of these cases, prediction points of $cfsb_1$ are the program points right after block b_3, i.e. Def_3 in auth=exp_0 right after block b_4, i.e. Def_4 in auth=... .

3) Indirect Control Flow Selector Variables (ICFSV) Algorithm: If each variable $v \in$ direct $CFSV$ is not control or data-dependent on any other variable, then the second static phase of CFI via DFI enforcement can be started after finding prediction points for each $cfsb$ in the program. Otherwise, in addition to variables in direct $CFSV$ that are directly used to select a path of the program CFG, there is a non-empty set of indirect $CFSV$ whose elements are variables that indirectly affect the runtime path selection. We provide, ICFSV in Alg. 3 in Section IV-B that returns the set of direct $CFSV$ and indirect $CFSV$ of the program.

B. ICFSV Algorithm

In the ICFSV Alg. 3, the set of direct $CFSV$ is computed as the union of all variables in $Use[b]$ for all blocks $CFSB$. Moreover, the algorithm starts with an empty set of indirect $CFSV$ and adds all variables that may indirectly affect the variables in direct $CFSV$. Details of this algorithm are explained in Section IV-B.

Assume that for each block b, the set of all blocks that b is data-dependent on them is denoted by $Data-Dep[b]$. Therefore, it is easy to see that the set of blocks that their definitions reaches and are used in b constitute the $Data-Dep[b]$. Therefore, $Data-Dep[b]$ can be computed according to the PRD algorithm because for each block this algorithm determines which variable definitions, including definitions of variables that are used in the block, reach it. This means, b uses variables that are defined in each block b' in $Data-Dep[b]$ while b' uses the set of other variables in $Used[b']$ to accomplish its definitions. Therefore, Alg. 3 takes $Data-Dep[b]$ for all blocks and adds the set of $Used[b']$ for each b' in $Data-Dep[b]$ to the set of indirect $CFSV$ by starting from a block, e.g. b, in $CFSB$. Assume that for each b, the set of all blocks that b is control-dependent on them is denoted by $Control-Dep[b]$. Using the program CFG, the reverse dominance frontier analysis algorithm in [17] of each basic block, computes all blocks that b has immediate control dependencies to them. Similarly, by starting from blocks b in $CFSB$, the Control Flow Selector Variable algorithm takes $Control-Dep[b]$ for each block and adds the set of $Used[b']$, for each b' in $Control-Dep[b]$ to the set of indirect $CFSV$.

This algorithm uses $Data-Dep[b]$ and $Control-Dep[b]$ to find all variables that there is a direct or indirect data or control dependency from them to any used variable in any $cfsb$. Therefore, the algorithm starts with adding used variables of each block to the set of indirect $CFSV$ if the block is in $TmpDepBlocks$ which is an initiated set by Data-Dep[cfsb] ∪ Control-Dep[cfsb]. Since the chain of these dependencies may include other blocks, the algorithm updates this list and continue

to find all variables in indirect $CFSV$. Also, to guarantee the algorithm termination, it is ensured that effects of variables in each block are considered at most once by marking processed blocks. Finally, the algorithm returns the set of all direct and indirect $cfsv$s of the program in $CFSVLst$.

C. CFI via DFI Instrumentation (Second Phase)

With the aim of monitoring program execution, the second phase of enforcement is started by initial steps which run analysis algorithms on the target programs. Afterwards, programs are instrumented in two different cases when indirect $CFSV$ is empty and when it is not.

Let $I(inst)$ denote the code that is added to the program through the specific code insertion called $inst$. Therefore, as respectively explained in this section, $I(SMaintaining)$ maintains and keeps track of true data flows for direct $CFSV$ and CF of program runtime paths. Then, $I(Prediction)$ predicts the $cfsb$ path selection in each $cfsb$ PP in each runtime path. In addition, before each $cfsb$, $I(Detection)$ checks whether the ongoing CF in the near future will follow the true path, and if not, $I(CFGuide)$ guides the runtime CF to the true path. If indirect $CFSV \neq \emptyset$, the last subsection explains extra enforcement steps. Briefly, in these extra steps, $I(SMaintaining)$ maintains and keeps track of true data flows for indirect $CFSV$ at runtime paths. In addition, $I(Detection)$ checks the truth of data flow for these variables. In case of detecting a data flow which is not true, $I(Restoration)$ rosters these variables to guarantee CFI via DFI satisfaction.

1) Initial Steps: Using Phoenix [29], we compile each source file to HIR, collect the target locations, and points-to constraints to a file. We implement the pointer analysis [18] which is an inter-procedural, flow and context insensitive analysis that scales to large programs and reads the constraint file, passes all source files to compute the points-to sets, which are locations that each pointer can point to, and stores them in another file.

During this analysis, we execute a safety analysis to determine unsafe variables and instructions to consider in the enforcement, reduce unnecessary instrumentation and thus runtime overhead of instrumented programs. In our safety analysis, an instruction is safe if it cannot illegally change any variable, and a variable is safe if there is no unsafe instruction that can modify it illegally. Similarly, a pointer that is dereferenced by any unsafe instruction is unsafe. Basic pointers such as frame pointer or data segment can be saved and manipulate in the protected part. However, instructions whose destination operand is temporary are considered safe when they modify a constant number of bytes by starting at a constant offset from these basic pointers. Moreover, during pointer analysis, we mark definition and use instructions, and mark unsafe target locations and instructions while we find $cfsb$s. If an unsafe storage is used in a $cfsb$, then it is an unsafe $cfsb$. To reduce the overhead, we consider unsafe $cfsb$s in the enforcement. Moreover, every block containing safe instructions is considered safe. To find safe definition instructions through pointers, we run an intra-procedural pointer-range analysis similar to [7] to check if definition instructions are always in bounds, and thus mark them safe. For example, in Fig. 1 instructions 1 is safe because it can only modify $auth$ while instruction 6 is not safe because it may modify other locations such as $auth$. Then, this variable, and thus the instruction using this variable to decide access grant or access deny is not safe. Therefore, we consider manipulation of this variable in the enforcement approach.

The next step in the initialization is based on the PRD algorithm and similar to [4], we use points-to sets to compute path reaching definitions by running the PRD algorithm for local variables that are just defined in the declared function and others when $IN[b]$ is also the union of the sets containing all definition to dereferences of pointers that may point to variables. Moreover, for pointer dereferences, this set is the union of the set containing the identifiers of all definition to the dereferenced pointer with the sets of all variables the pointer can point to.

Using $IN[b]$ and $OUT[b]$ as the results of the PRD algorithm, we run CFSPP algorithm to find $DefSeq$s and prediction points for unsafe $cfsb$s. Finally, to compute indirect $CFSV$, we execute the ICFSV algorithm and consider its unsafe subset in the approach. The results of these algorithms are written to a file for the next phases.

2) State Maintaining, Restoration and Prediction Instrumentation: Let us before the explanation of our policy enforcement, do a top-down review of the policy. According to the CFI via DFI Policy, the path selection in each $cfsb \in CFSB$ of a program in each runtime path should correspond its predicted one which is determined in the $cfsb.PredPoint$ or shortly PP in the RP. Considering indirect $CFSV = \emptyset$, the $PredPoint$ for $cfsb \in CFSB$ of a program with the pair of ($cfsb.DefSeq$, $cfsb$) in the runtime path RP is right after the last definition instruction of $cfsb.DefSeq$ in that path if for each data flow $Def \in cfsb.DefSeq$ to the $cfsb$ DFI satisfaction condition is satisfied. To make sure about the satisfaction of this condition in our enforcement approach, the *state maintaining* and the *restoration* instrumentations are proposed that respectively maintain the legally defined values in the protected part and restore them to use without any possible unauthorized changes.

To reduce the overhead of the state maintaining and the restoration instrumentations, we also distinguish between $DefSeq$s. To do so, in the initialization steps, we check if definitions in a $DefSeq$ of an unsafe $cfsb$s are in consequent safe blocks or not. In a consequent $DefSeq$ there is no unsafe instruction, and thus because of NOC assumption, there is no illegal modification in defined variables in the sub-path starting from their definition to the PP. The state maintaining instrumentation adds $I(SMaintaining)$ immediately after each Def in each non-consequent $cfsb.DefSeq$, except for the last definition that with no intermediary precedes the PP and thus can not be modified illegally, to keep track of true data flows. $I(SMaintaining)$ for each $Def(v)$ saves the assigned value to variable v in the protected part by a new high-level instruction with the following form when `identifier` is the variable name and `val` is its value:

TABLE III
BUFFER OVERFLOW TAXONOMY ATTRIBUTES OF PREVENTED ATTACKS BY CFI VIA DFI ENFORCEMENT.

Attribute name	**Attribute value**
Memory Location: where the overflowed buffer resides	Stack, Heap, Data region, BSS
Scope: scope difference between buffer allocation and overflow	Same scope, Inter-procedural, Global
index complexity: the complexity of the array index	Constant, Variable, Linear expression, Nonlinear expression, Function return value, Array contents
Alias of Buffer Index: whether or not the index is aliased	No , One level, Two levels
Local Control Flow: what kind of program control flow most immediately surrounds or affects the overflow	None, If, Switch, Goto/label
Secondary Control Flow: either precedes the overflow, or contains nested local control flow that affects the overflow	None, If, Switch, Goto/label
Loop Structure: describes the type of loop construct within which the overflow occurs	None, Standard for, Standard do while, Standard while, Nonstandard for, Nonstandard do while, Nonstandard while
Loop Complexity: indicates how many loop components from initialization, test, increment are more complex than initializing to a constant, testing against a constant, and incrementing or decrementing by one	No loop, None, One, Two, Three
Container: type of container the buffer is within	No container, Structure, Union, Array
Data Type: what type of data is stored in the buffer	Character, Integer, Floating point, Pointer, Unsigned integer
Signed/unsigned type error: the buffer overflow is caused by a signed vs. unsigned type mismatch, No	Yes
Overflow Magnitude: the size of the overflow	Limited, Non limited
Continuous/Discrete: whether the overflow is discrete or continuous	Continuous, Discrete
Upper/Lower: which buffer bound gets violated	Upper, Lower
Runtime Environment Dependence: whether or not the occurrence of the overrun depends on something determined at runtime	Yes, No

`MAINTAIN identifier val`

If a definition of a variable belongs to several $DefSeq$s, it is instrumented once by $I(SMaintaining)$. Therefore, the instruction identifier of each instrumented definition is saved in a file and used in the process of the enforcement. Fig. 4(a) shows how this instrumentation changes each $cfsb.DefSeq$.

After the state maintaining instrumentation, the program in all prediction points of each $cfsb$ is instrumented by $I(Prediction)$. In $I(Prediction)$, the $cfse$ of the $cfsb$ is predicted by restoring the value of every maintained v that $Use(v) \in cfsb.ML$ from the protected part through another high-level instruction, denoted by $I(Restoration)$, with the following form with the similar parameters as `MAINTAIN`:

`RSTR identifier val`

As shown in Fig. 4(a), for $cfsb$ prediction it is precomputed in a temporary variable denoted by tmp. Therefore, since the legal value of required data is maintained in the protected part and restored from it, DFI satisfaction condition is guaranteed for $cfsb$ prediction. Then, the precomputed $cfse$ value is saved in the protected part by $I(SMaintaining)$ which is added immediately after $I(Prediction)$ through `MAINTAIN cfsb-id tmp` instruction when `cfsb-id` is the identifier of the $cfsb$ and `tmp` is its value. Therefore, the expected $cfse$ value is saved in the protected part for future use to guide the runtime CF to the true path.

V. PROGRAM STATE

The execution of instructions along a program runtime path with the aim of analysis can be viewed as program state transitions [1]. Each program state may consist of the program data memory, information about the stack, code memory and etc. For every instruction, the associated state with the program point before the instruction and after it are called the instruction input and output states, respectively [1]. The execution of each instruction transforms this input state into the output state. In addition, operational semantics for instructions determines how they affect program execution states [24]. Operational semantics for an instruction such as $I1$ can be shown as $s_1 \xrightarrow{I1} s_2$ where s_1, and s_2 are the instructions input and output states, respectively. In this paper, the program state s consists of program data memory M_d and protected data memory M_{PD}. For example, according to NOC and programs state transitions, if $I1$ and $I2$ with state transitions $s_{i-1} \xrightarrow{I1} s_i$ and $s'_i \xrightarrow{I2} s_{i+1}$ are two sequential program instructions, then $s'_i = s_i$.

If the program state before an instruction `MAINTAIN` is s_i and its state transition is $s_i \longrightarrow s_{i+1}$, then $s_{i+1}.M_d = s_i.M_d$, and $s_{i+1}.M_{PD}(v) := s_i.M_d(v)$, i.e. $s_{i+1}.M_{PD}(v)$ is assigned the value of $s_i.M_d(v)$. Let us consider the same state transition for `RSTR`, it is clear that this instruction does not make any changes in M_{PD} while $s_{i+1}.M_d(v) := s_i.M_{PD}(v)$. Moreover, these instructions are assumed to be free from any vulnerabilities.

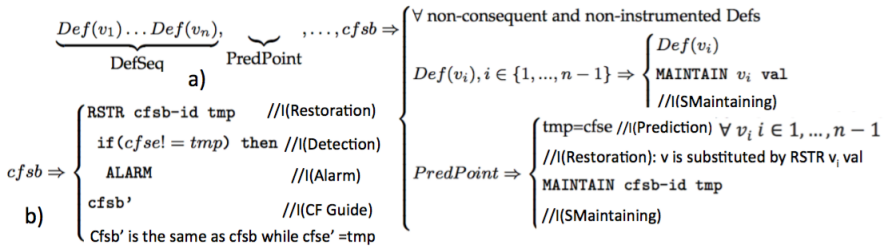


Fig. 4. Conversion for unsafe *cfsb*s through a) State Maintaining and Prediction Instrumentation, and b) Detection and Control Flow Guide Instrumentation.

```
      _auth = ASSIGN 0                          #1
      _tmp =CMP(NE) _auth, 0        //I(Prediction)
      MAINTAIN 6 _tmp             //I(SMaintaining)
$L6:   t274 =CMP(NE) _exp, 0                    #2
      CBRANCH(NE) t274, $L8, $L7                #3
$L8:       _auth = ASSIGN ...                   #4
      _tmp =CMP(NE) _auth, 0        //I(Prediction)
      MAINTAIN 6 _tmp             //I(SMaintaining)
$L7:   tv275 = CALL &_Strfunc, &_buffer         #5
      RSTR 6 _tmp                        //I(Restoration)
      t279 =CMP(NE) _auth, 0                    #6
      t280 =CMP(NE) _tmp, t279       //I(Detection)
      CBRANCH(NE) t280, $L13, $L14
      $L14: tv279 = CALL _ALARM       //I(Alarm)
      $L13: CBRANCH(NE) _tmp, $L11, $L10        #7
//I(CF Guide)
$L11: tv277 = CALL &_access-grant &_unam        #8
      GOTO $L12                                 #9
$L10: tv278 = CALL &_access-deny &_unam         #10
$L12:
```

Fig. 5. Instrumented form of the example vulnerable code in Fig. 1.

To depict an example of the enforcement, Fig. 1(b) shows the HIR form of the vulnerable fragment code, and Fig. 5 shows its instrumented form.

To maintain values of *cfsb*s expressions, we allocate the first 100k of the protected part, and thus the address of target locations in the protected part will be after 1A00h. Therefore, instruction `MAINTAIN identifier val` is lowered as follows where [*_identifier*] refers to the identifier address and *_identifier* refers to its value:

```
lea ecx,[_identifier]
mov word ptr [fs:ecx+1A00h],_identifier
```

At first in this instruction, the target address of the definition instruction is loaded into `ecx`, and using the FS by adding the 4K guard page and 100k for *cfsb*s, it is used to update the protected part to maintain the new value. Instruction `RSTR` is lowered similarly, except that in the `mov` instruction the parameters are in the revers order. Moreover, instruction `MAINTAIN cfsb-id tmp` is lowered as follows, and the parameters of `mov` appear in the reverse order for instruction `RSTR cfsb-id tmp`:

```
lea eax,[_cfsb-id]
mov word ptr [fs:eax+1000h],_tmp
```

1) Detection and Guide Instrumentation: After prediction of each unsafe *cfsb*, to guarantee that the runtime CF of the program takes the true runtime path from the CFI via DFI point of view, the enforcement approach in detection instrumentation step adds $I(Detection)$ right before each *cfsb*. According to the previous instrumentation, the true value of the $cfse$ for each *cfsb* is predicted and saved in the protected part which is denoted by $cfse.tmp$. For detection, first $I(Restoration)$ restores $cfse.tmp$ from the protected part through `RSTR cfsb-id tmp` instruction when `cfsb-id` is the identifier of the *cfsb* and `tmp` is $cfse.tmp$ value. Then, $I(Detection)$ checks the correspondence between $cfse.tmp$ and its current value that is going to be used to select the ongoing path. If these two values are the same, then the runtime CF goes on. Otherwise, an imminent CFI via DFI violation is detected and an alarm is raised through $I(Alarm)$. $I(Alarm)$ is a simple instruction without any vulnerability that informs the user of a detected violation. Then, the CF is guided to the true path through $I(CFGuide)$ as if no violation was going to happen. $I(CFGuide)$ is the same code fragment as *cfsb* except that instead of the current value of the $cfse$, it uses $cfse.tmp$ from protected part to select the ongoing path. Fig. 4(b) shows all conversions through theses instrumentation steps. It is obvious that the detection instrumentation uses each variable v that $Use(v) \in$ *cfsb.ML*, and thus is marked as $I_{Det}\,(Use)\,(v)$. Moreover, $I(Alarm)$ do not use or define any variable in the program data memory while $I(CFGuide)$ is marked as $I_{CFGd}(Use(tmp))$. An example of the enforcement in this phase for the vulnerable code fragment in Fig. 1, is shown in Fig. 5.

2) Enforcement with Indirect Control Flow Selector Variables: If there is a non-empty set of indirect $CFSV$, according to the definition of prediction point, DFI satisfaction condition should be satisfied for all data flows in the RP preceding the $cfsb.DefSeq$ related to each variable $v \in$ indirect $CFSV$. To dynamically guarantee satisfaction of this condition, for each

$$Def(v) \Rightarrow \begin{cases} Def(v) \\ \texttt{MAINTAIN v val } Use(v) \\ \texttt{//I(SMaintaining)} \end{cases} \quad Use(v) \Rightarrow \begin{cases} \texttt{RSTR v tmp} & \texttt{//I(Restoration)} \\ \texttt{if}(v! = tmp)\ \texttt{then} & \texttt{//I(Detection)} \\ \quad \texttt{ALARM} & \texttt{//I(Alarm)} \\ \quad v = tmp & \texttt{//I(Recovery)} \\ Use(v) \end{cases}$$

Fig. 6. *df* : $Def(v) \rightarrow Use(v) \in DFG$ conversions through instrumentation steps for every unsafe $v \in$ indirect $CFSV$.

data flow including definitions and uses of each unsafe variable $v \in$ indirect $CFSV$, $I(SMaintaining)$ is added after each definition of v to save its last value into the protected part. In addition, before each instruction is marked as $Use(v)$ for such a variable, $I(Restoration)$ and $I(Detection)$ are added to check the truth of data flow for that variable. By $I(Detection)$, the correspondence of the current value of v with its saved value is checked. If these two values do not correspond to each other, a data flow which is not true is detected. Moreover, the enforcement approach can instrument the program by $I(Alarm)$ to inform the user about a violation from DFI which also may lead to a violation from CFI via DFI. Furthermore, the enforcement approach has a recovery instrumentation step added right after $I(Alarm)$ that in case of such a violation, recovers the current value of v. To do so, we again use `RSTR` instruction. which is a definition for variable v, and thus is marked as $I_{Rec}\,(Def)\,(v)$.

3) Enforcement Complementary: To complement the enforcement, we use the similar technique for control-data introduced in the compilation process such as function calls and return. For example, we maintain the return address on function entry, we load the `esp` to the `ecx` and save the return address in the protected part while the destination address is computed as other target addresses as the following:

```
lea ecx,esp
mov word ptr [fs:ecx+1A00h],_identifier
```

To restore the return function to tmp, we perform the `mov` instruction and use it to check its integrity as the following:

```
lea ecx,esp
mov word ptr,_tmp,[fs:ecx+1A00h]
```

Indirect calls that depending on the platform can be accessed through jump tables or others are placed in the protected memory, and thus cannot be overwritten by any code except for the enforcement code. In addition to that, the value assignment to the function pointers and their use are treated as indirect control flow selector variables. We also provide some function wrappers that use and define unsafe and critical data to call functions that are not instrumented in our approach. These wrappers take the same arguments as the function in addition to the legitimate value of data while its last legal value is maintained, and we change the function calls to call wrappers. Moreover, before the function call in the wrapper, we check if the value of such data is correct, and after the function call we update the last value of defined data in the function. Different platforms such as OS with their specific executable file format, and the underlying language affect compiled code. For example, the compiled code may include some constructors (initialization functions) which need to be called before the program execution starts (e.g. before the main is called), or some destructors (termination functions) that should be called when the program terminates. However, the list of initialization and termination function pointers are generated by compiler that traditionally are known as âĂŸ.ctorsâĂŹ and âĂŸ.dtorsâĂŹ. Then such lists will be traversed at startup time and exit time. These lists can be stored in the protected part and then be traversed later. In our case, we locate these lists in the protected part and change libgcc2.c to traverse them from the protected part.

To support static and dynamic linking of compiled modules, we consider a rare but a possible situation in our first phase of the enforcement. Such a situation may happen when a variable (may be a global variable) is a non-control-data defined in one module while it is a control-data in another one. Therefore, we extend our static analysis by considering and determining the following three sets as auxiliary information for each module:

- $AOUT$ (All OUT going data), as a bit vector of all definitions, including all control and non-control-data that reach the module exit,
- $ContD$ as a boolean bit vector of all definitions setting to 1 and 0 indicating the corresponding definition is control-data or not in the module,
- $CdIN$ (Input Control-Data), as a bit vector of control-data that are not defined in the module but are used in at least one $cfsi$. To determine this set, we insert dummy definitions at the beginning of each module for all used variables in $cfsi$s and check their propagation by running our static analysis algorithms. If one of these definitions can reach a $cfsi$, then the variable is in $CdIN$ of the module.

Afterwards, the second phase of the enforcement which is the instrumentation is applied on each module. In addition to the usual instrumentation such as I(SMaintaining) for each control-data definition, all non-control-data definitions in that module will also be instrumented conditionally. It means that for a definition such Def_i which is not a control-data in module m, the instrumentation is "`If` $ContD[m][i]$ `then` $I(SMaintaining)$". Then, for each module the auxiliary information is maintained in the protected part. Moreover, all $cfsi$s in each module that do not use any variable in $CdIN$ are instrumented as explained in the enforcement with no change (i.e. their prediction points will be found and etc.), and the instrumentation for the other $cfsi$s can be done right before themselves in the code. When modules are linking together, auxiliary information should be

combined and updated. However, depending on the platform the process of linking is a little bit different. Executable and DLL loading in Windows is similar to loading a dynamically linked program in Linux. However, a difference is that in Windows the linker is a part of the kernel, and thus the kernel maps in the executable guided by the Portable Executable File Format (PE) header. Then the loader maps all dependent modules and updates the Import Address Table (IAT) based on the export tables [33]. Therefore, we instrument the loader to maintain the IAT and the export tables in the protected part. Moreover, the loader is changed to update the IAT in the protected part during its process. In addition, we instrument the loader itself similar to other code. When it is desired to map module m_i after m_j, for variables in $(AOUT[m_j] \bigcap CdIN[m_i])$, the loader should change the $ContD[m_i]$ to true as well. Moreover, for the resulted module m, there are three adjustments:

$$AOUT[m] = AOUT[m_i],$$
$$ContD[m] = ContD[m_j] \cup (AOUT[m_j] \bigcap CdIN[m_i]),$$
$$CdIN[m] = CdIN[m_j].$$

Using the above information modules can be linked together. However, for more precise module linking, one may include other auxiliary information for each module, such as type information in MCFI [32]. In addition, in Linux some data structures such as Global Offset Table (GOT) and Procedure Linkage Table (PLT) need to be maintained in the protected part and dynamically be adjusted by the instrumented dynamic linker. For example, the GOT entries should be maintained in the protected part and be adjusted from the address of the linker to the addresses of corresponding library functions. To support separately compiled modules in Linux and adjustment of these data structures MCFI [32] provides a useful approach.

In multi-thread settings, each thread has its own space in the memory and thus in the protected part for its local variables. Moreover, shared variables between thread have their own synchronization which is also enough for instrumentation. Considering dynamic linking, concurrent access to the same auxiliary information of a module might happen. For example, one thread may dynamically load a module, and thus need to update the auxiliary information while another thread may need to access them concurrently. Moreover, some data structures such as IAT or GOT can be accessed simultaneously by different threads. We provide a critical section for the auxiliary information or those table manipulations to prevent such interferences. To do so, we used the possibility of `CRITICAL_SECTION` which are lock objects with a pseudo code such as the following:

```
CRITICAL_SECTION cs;
EnterCriticalSection(&cs);
// access critical section
LeaveCriticalSection(&cs);
```

Since in SPEC 2006 benchmark, DLLs are loaded at the beginning of the program execution, these interferences are prevented. However, there is a possibility to load and unload libraries at the middle of execution for example in Java but it is a rare event [32]. We also consider Just-In-Time (JIT) compilation and binary rewriting as our future work to prevent existing attacks [27], and attacks related to context switching in [28] specially by considering commercial software with less accessible information of the code [26].

As explained based on allocated memories, to prevent unexpected changes in protected part we check if the bitwise logical `AND` of FS with the address of an unsafe definition is non-zero as follows:

```
lea ecx,address
test ecx,fs
je L
int 3
L:
```

4) Summary: Summarizing the CFI via DFI enforcement approach, we explain how applying this approach affects programs runtime paths and how it guarantees satisfaction of the policy by a Lemma.

VI. Summary and Proof of Lemma

In this section, we Summarize the CFI via DFI enforcement approach, and explain how applying this approach affects programs runtime paths and how it guarantees satisfaction of the policy by a Lemma.

The data flow analysis including the reaching definition algorithm is conservative, and the points-to analysis is an over-approximate analysis. Therefore, based on these two analyses, for a program code, an over-approximate sets of all variables that a) might be defined by an instruction, and b) might be reached and used in an instruction are derived. The conservation and over-approximation properties in these static analyses together with dynamic checking cause by IRM prevent false positive in related work such as DFI, WIT and ours as well. However, the policy is not dependent on these two static analysis. For example, a pointer analysis proposed and used in [30] can be used instead of the point-to analysis. In contrast, using static analysis to determine not only the insertion points but also what to check dynamically at these points, which is used and explained in details (Section VII) such as [4], [7], leads to some false negatives. Instead, in this paper we just use the static analysis to find the insertion points, and instead of tracking instructions which are making changes, we track the legally-assigned values of data at runtime which are determined dynamically, and thus prevent those false negatives. Besides, instructions that are considered safe, and thus have not been instrumented may cause false negative in previous work. If somehow such instructions make an unauthorized change in data, these approaches do not detect them. Although CFI via DFI does not impose extra overhead to instrument safe instructions as well, because of checking the current and the last legally-assigned value of data in its usage, this

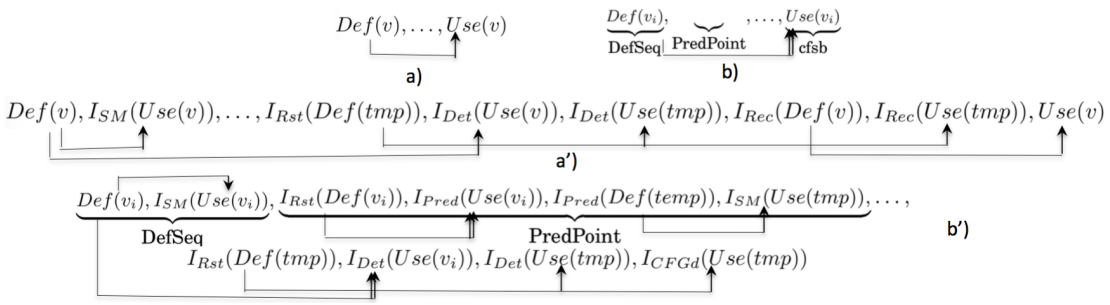


Fig. 7. The sequence of instructions in original programs in a) for $v \in$ indirect $CFSV$ and b) for $v \in$ direct $CFSV$, and the sequence of instructions in the instrumented program respectively in a') for $v \in$ indirect $CFSV$ and b') for $v \in$ direct $CFSV$.

kind of false negative is also prevented. Similarly, this characteristic of our approach can also prevent false negative related to attacks with the aim of tampering with the enforcement code or target addresses of indirect control transfers, by checking the legal value in address-control-data. However, there is some limitations and false negatives common between all these works and ours which is caused by the used field-insensitive analysis, that is, these approaches do not distinguish between the different fields in the same structure. Therefore, if an instruction that is just authorized to change a field of a structure, changes another field, these approaches do not detect. Using a field-sensitive points-to data flow analysis can be considered as a future work. As described in details, other limitations in the static analysis such as global variables or communications between different modules are considered by extending the static analysis. Another benefit of our approach is that, since instead of tracing instructions that make data changes, it traces data value changes, after an imminent CFI via DFI violation detection it recovers data, and the program execution continued as if no violation was going to happen.

Lemma 1 (Enforcement)**.** *If the CFI via DFI enforcement approach is applied to a program, the CFI via DFI policy is satisfied.*

Proof. this proof that shows that the effects of analysis approach aside, our proposed approach does not have any false positive and has just the false negative based on the used analysis technique. Moreover, the approach not only detects a violation, but it also makes the program continue its execution because of supporting a recovery technique.

Based on semantics of instrumentation, `MAINTAIN` in $I(SMaintaining)$ is marked as $I_{SM}\,(Use)\,(v)$ because this instruction just uses v of the program data memory to save its value to a location in the protected part. By a similar explanation, $I(Restoration)$ is marked as $I_{Rst}(Def(v))$ for every restored v, and $I(Prediction)$ is marked as $I_{Pred}(Def(tmp))$. Now, let us look more specifically to the CFI via DFI enforcement approach that converts each runtime path RP to an Instrumented Runtime Path (IRP). For example, considering each $v \in$ direct $CFSV$, definitions in non-consequent blocks in $DefSeq$ except for the last definition in the RP for each *cfsb* such that $Use(v) \in cfsb.ML$, IRP includes $Def(v)$, $I(SMaintaining)$ which according to their marks are shown as $Def(v)$, $I_{SM}(Use(v))$. With similar explanation, Fig. 7 shows RP and IRP. To make the following and reasoning as simple as possible, the sub-paths of RP and IRP for $v \in$ indirect $CFSV$ in a pair of $Def(v), ..., Use(v)$ are shown in Fig. 7(a) and Fig. 7(a'). Moreover, Fig. 7(b) and Fig. 7(b') show sub-paths in RP and IRP for a pair of ($cfsb.DefSeq$, $cfsb$) which manipulate $v \in$ direct $CFSV$. In this figure, a data flow is shown with an arrow while multiple data flows are shown by duplicate arrows at their ends. As shown in Fig. 7(a) and Fig. 7(b), there are data flows which their definition and use might be far from each other and also they might be broken by unauthorized definition for the data between their legal definition and use. However, as shown in Fig. 7(a') and Fig. 7(b'), the enforcement approach defines some new local data flows by instrumentation, and change the existing data flows $Def(v), \ldots, Use(v)$ to $Def(v), \ldots, I_{Det}(Use(v))$ for v in Fig. 7(a'), and v_i in Fig. 7(b'). These new data flow are generated in the worst case of a violation occurrences by an unauthorized data change, e.g. in Fig. 7(a'), and when definitions are in non-consequent safe blocks shown in Fig. 7(b) and Fig. 7(b'). Otherwise, the inserted data flows are decreased. For example, in a case of no violation in Fig. 7(a) and Fig. 7(a'), the two data flows caused by recovery would be omitted and in a case of consequent definitions in Fig. 7(b) and Fig. 7(b'), the IRP would not have the two data flows with multiple arrows related to the I_{SM} and I_{Rst}. In addition, for the simplicity, we ignore exceptions such as the instrumentation for the last definition $DefSeq$s.

To prove the lemma, the following IRM requirements [8] must be satisfied:

1) Integrity: The secured application must not be able to circumvent or subvert the IRM enforcement code.
2) Transparency: In the absence of the policy violation, observable behavior of secured application must not be modified.
3) Soundness: The observable executions of the secured application by IRM, i.e. IRPs must satisfy the policy.

Based on these requirements, if the integrity and transparency are satisfied, then the IRP corresponding to an RP with no violation, would satisfy the policy, i.e. there is not any false positive. Moreover, the effects of analysis approach aside, if soundness is satisfied, then any violation in an RP must be detected in its corresponding IRP, and thus the approach do not have any false negative.

By checking the target addresses of writes and control transfers, tampering with the protected part and bypassing the IRM is prevented, and thus the integrity requirement is satisfied. Moreover, since the explained enforcement code does not have any vulnerability against the desired policy, if an RP does not violate the policy, the instrumentation does not cause any policy violation. In addition, soundness is satisfied because IRPs satisfy CFI via DFI. Otherwise, by using the contradiction, according to the definition of this policy, at least there must be one $cfsb \in CFSB$ of a program in the IRP that its path selection does not correspond its predicted one. In such a case, CFI via DFI violation might be a result of a wrong prediction or inconsistency between true prediction and ongoing path selection. Considering a violation of the CFI via DFI as a result of an unauthorized change in a data by an arbitrary instruction that might be anywhere in the shown sub-paths, these two possibilities imply three different cases as follows:

Case 1: DFI is not satisfied in IRP preceding the $cfsb.DefSeq$ for a variable $v' \in$ indirect $CFSV$ which a used variable in the $cfsb$ is dependent on,

Case 2: DFI is not satisfied for data flows from $cfsb.DefSeq$ to $cfsb.PredPoint$ in IRP for a variable $v \in$ direct $CFSV$ where $Def(v) \in cfsb.DefSeq$ and $Use(v) \in cfsb.ML$, or

Case 3: actual path selection does not correspond to the true predicted one.

The first two cases result in a wrong prediction for the $cfsb$ and are not possible situations. It is because if an unauthorized change in a variable $v \in$ indirect $CFSV$ can cause a DFI violation, then there is a runtime data flow between the $Def(v)$ and $Use(v)$ that is not a true data flow. As shown in the figure a'), the first use of v happens in $I_{Det}(Use(v))$ which is in detection instrumentation and to compare its current with its saved value. Then, in a case of an unauthorized definition to the variable, it is recovered and thus the DFI satisfaction condition holds after its original definition starting the sub-path in figure a') and before its actual use ending this sub-path. Therefore, case 1 could not be possible. For the second case, since for each $v \in$ direct $CFSV$ where $Def(v) \in cfsb.DefSeq$ and $Use(v) \in cfsb.ML$, the legal value is maintained, and in the prediction point is used, then similarly even in case of a violation, the DFI satisfaction condition holds from the $DefSeq$ to the prediction point of each $cfsb$. Therefore, prediction can not be wrong. Moreover, the actual path selection will correspond to the true predicted path because the first use of variables after prediction happens in I_{Det} right before the $cfsb$ to compare the current value of the $cfse$ with its predicted one. Since predicted amount of the $cfse$ is saved and restored from the protected part, the DFI satisfaction condition is preserved. Therefore, any CFI via DFI violation before the actual path selection is detected. In addition according to $I(CFGuide)$, $cfsb'$ uses the true predicted value of $cfse$ to select the ongoing path. Hence, case 3 can not be true as well. □

VII. Related Work

Many control-data attacks which according to [9] are among the most critical security attacks result in subverting programs runtime CF [2], [4], [6]. Therefore, indirect or direct attention to CFI, e.g. in [2], [10] and when it is affected by DFI, e.g. in [4], [14], [30] is not new. However, in previous work, mostly address-control-data was considered as control-data, and attacks through non-address-control-data which subtlety can affect the integrity of runtime CF still remain untreated [4], [6]. There is useful work such as [14], [7], [4] that can indirectly prevent these attacks but they impose high overhead and false negatives.

CFI was firstly defined in [2] with an enforcement that makes many exploits unsuccessful. This work uses a machine-code rewriting that instruments programs with runtime checks. It dynamically ensures that indirect calls in the CF remain within a given CFG by setting and checking unique labels in the source and target of the control transfer during indirect calls. Moreover, indirect control-transfer is also taken into account by some other work such as CPI [30], or CCFIR [20] to validate a target more simply and faster than traditional CFI, and in [21] for COST binaries. Program Shepherding [10] is another security enforcement mechanism that prevents many attacks against CFI but employs a machine code interpreter with remarkable overhead. Although these methods aim to prevent some kinds of CFI violations and provide a generic defense against address-control-data attacks, they cannot defend against non-address-control-data attacks [4], [7].

DFI was firstly defined in [4] informally; however, its specification is analysis-dependent, based on reaching definition, and is not comprehensive [14]. DFI enforcement in [4] instruments the program to check the identifier of definition instruction of a variable be in the set of reaching definitions for every use of that variable. The memory safety approach presented in WIT [7] is another useful technique to satisfy DFI which uses static data flow analysis to assign a color to each object and to each definition instruction such that each instruction and all objects that can be defined by it have the same color. WIT generates instrumented programs that record object colors at runtime and check that instructions define objects with the same color. Moreover, a comprehensive and analysis independent definition of DFI with foundations for its enforcement techniques is presented in [14]. In its proposed technique, static analysis is used to find CFG and DFG of programs, and based on some provided dynamic rules, programs are instrumented to check if not only every data definition but also every data use in a runtime path are legal and make a true data flow. However, in the specification and enforcement of CFI via DFI, we introduce the *DFI satisfaction condition* property of true data flows based on DFI specification in [14].

One of the main differences of CFI via DFI with previous work is that their goal is a wider issue which is memory safety, e.g. [7], or DFI, e.g. [14], [4] for all categories of data and not just control-data, and thus their enforcement techniques come with significantly higher overhead. In addition, in these methods, static analysis is used not only to determine insertion points

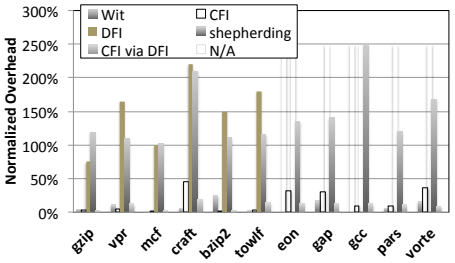


Fig. 8. Execution overhead of CFI via DFI enforcement on SPEC 2000 benchmarks in comparison with the overhead of the related techniques.

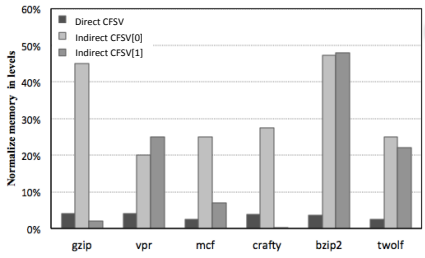


Fig. 9. Ratio of leveled target variables for CFI via DFI attacks in commonly choosed programs of SPEC 2000 benchmarks.

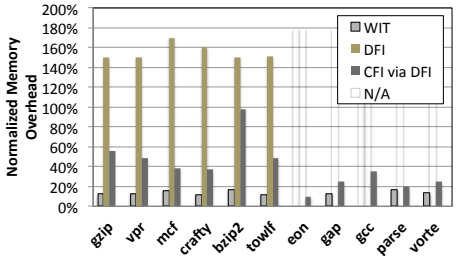


Fig. 10. Space overhead of the CFI via DFI enforcement relative to the execution without instrumentation on SPEC 2000 benchmarks in comparison with the available overhead of the related techniques.

but also to determine what to check dynamically at the insertion points which are statically determined colors, in WIT, and identifier of instructions in [4]. This leads to some false negatives. Instead, in this paper we just use the static analysis to find the insertion points, and instead of tracking instructions which are making changes, we track the legally assigned values of data at runtime which are determined dynamically, and thus prevent those false negatives. For example, in DFI, reaching a data from the set of all definitions that according to a conservative static analysis may reach a use at runtime, is considered true. For instance, regardless to exp in Fig. 1, both of definitions in line 1 and 4 are in the set of reaching definition computed by the analysis for the instruction in line 8. However, if exp is always false then only definition in line 1 must reach the use in line 8 while even reaching the definition in line 4 is considered true at runtime. Therefore, an attacker can not change the $auth$ value and claim that its value is coming from line 4 and continue undetected because the variable is checked to hold its last legal assigned value in line 1. Similarly, in WIT defining a data by an instruction that just might be defined in a specific situation is always considered legal because of having the same color which leads to false negative. However, [14] does not have this category of false negatives because of considering current runtime path, while due to more independent and general specification and enforcement of DFI, imposes a higher overhead than [7], [4].

Another category of false negative in previous work is related to the instructions that are considered safe, and thus have not been instrumented. If somehow such instructions make an unauthorized change in data, these approaches do not detect them. Although CFI via DFI does not impose extra overhead to instrument safe instructions, and thus those checks are not run by instrumented programs at runtime, because of checking the value of data it does not have such kind of false negative. In addition, based on the part of the enforcement that aim to prevent tampering with the enforcement code or target addresses of indirect control transfers, the previous works have another category of false negative. DFI uses the proposed approaches in [2], [10], and WIT uses its own approach to do that. Regardless to the used approaches, the attacker might be able to arbitrary transfer the control between objects with the same statically assigned label as in [2] or the same color as in [7]. Similarly, by checking the legal value in address-control-data in the CFI via DFI approach, this kind of false negatives are prevented. However, false negatives caused by the used field-insensitive analysis are still common between all these works and ours. Another difference between ours and the previous approaches is that, they prevent or detect violations while none of them can recover the data. The enforcement approach in this paper detects imminent CFI via DFI violations and prevents them. Moreover, since instead of tracing instructions that make data changes, it traces data value changes, CFI via DFI enforcement recovers data, and the program execution continued as if no violation was going to happen. In the explained related work with the aim of DFI detection or prevention, there is not any false positive. However, it is clear that due to considering data integrity including non-control-data, they have remarkable false positive in detection of CFI via DFI violations. In our approach, when a DFI violation related to an unauthorized change in indirect $CFSV$ is detected, it will be recovered, and an alarm is raised to inform the user of a possibility of a CFI via DFI violation in the ongoing execution. It is because the runtime CF containing this violation might pass through a CFG path of the program that does not include manipulation to direct $CFSV$ and $CFSB$s which might be affected by the violation. However, alarm instrumentation is optional and just recovery instrumentation can be sufficient to prevent CFI via DFI attacks.

VIII. Evaluation

To evaluate the CFI via DFI policy enforcement, we measured its runtime and memory overhead by applying the enforcement on programs in the CPU intensive SPEC 2006, and SPEC 2000 [15] benchmarks with extra analysis on programs of the SPEC 2000 benchmark that are common between the most related work to discuss about a possible optimization. Moreover, we checked the effectiveness of the approach against some real and synthetic security attacks. Our experiments show that the approach achieves reasonable performance overhead in addition to a remarkable security attack covering in both of the real and synthetic attacks.

Execution time and memory overhead of the related techniques are measured on different platforms, with various assumptions, to satisfy different policies when their approaches are applied on different set of programs. For example, the platform in a more related work such as [4] is an idle Dell Precision Workstation 350 with a 3GHz Intel Pentium 4 processor and 2GB of memory, and in [7] is an idle Dell Optilex 745 Workstation with a 2.46GHz Intel Core 2 processor and 3GB of memory. In addition, both of these techniques and other explained works are applied on programs of the integer benchmarks in SPEC 2000

[15] running on Windows. However, to perform overhead measurements of the CFI via DFI enforcement while its comparison with those techniques is facilitated as much as possible, we chose a platform close to those techniques and performed our experiments on a Dell Vostro 1500 with Intel Core 2 Duo processor at 2.5GHz and 2GB of RAM, when programs are compiled using gcc -O3 on Windows XP SP2.

The normalized overhead of the CFI via DFI enforcement, which is the ratio of the overhead of execution time of each benchmark after enforcement to the execution time of the original SPEC 2000 benchmark, is shown in Fig. 8. In this figure, the overhead of our approach, denoted by CFI via DFI, is accompanied by the available overhead of other approaches in [2], [7], [4] and [10] denoted by CFI, WIT, DFI, and Shepherding keys, respectively. Since those techniques are not applied to all programs in SPEC 2000, in this figure some columns are shown as not applicable (N/A). As depicted in the figure, due to detecting address-control-data attacks, CFI and shepherding result in lower overhead than DFI which can detect data attacks. The overhead of CFI via DFI is between the lower and higher overhead of related work while it can detect and recover control-data in a case of a violation. Furthermore, CFI via DFI incurs its highest overhead in crafty which is significantly lower than the highest overhead in most of the others. It is because the ratio of the set of control-data that its manipulation is controlled by the approach, to all data in crafty, shown in Fig. 9, is considerably small. Moreover, our experiment shows that the memory and runtime overhead introduced because of static and dynamic linking is low. For example, in SPEC 2006 the memory overhead is between 4.5% to 6%, and runtime overhead is negligible due to their low ratio of dynamic linking events in comparison to other considered events in the approach ($\ll 10^{-8}$) [32].

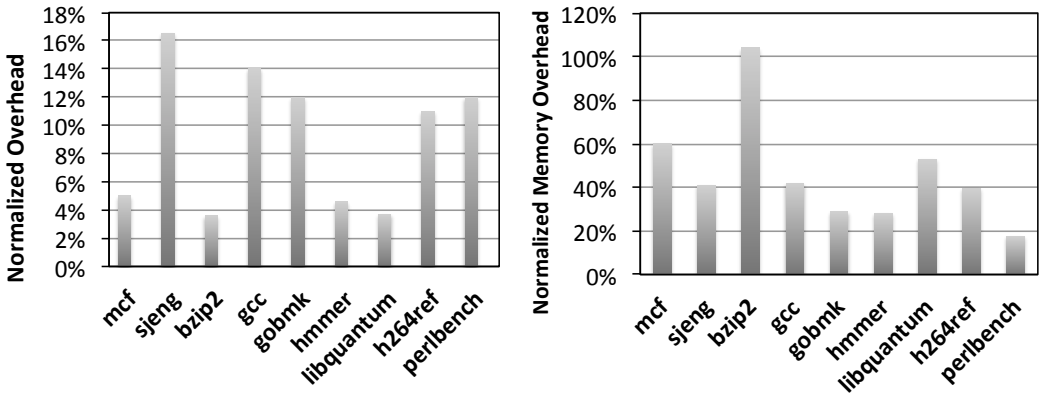
Fig. 9 shows the ratio of direct $CFSV$, and indirect $CFSV$ in two levels for common programs of SPEC 2000 benchmark the most related work are applied on, and although we did not check this ratio for others, we do not find any evidence that might make an exception in other programs. It is obvious from the figure that in all these programs, direct $CFSV$ is significantly smaller than the other categories. Furthermore, except for bzip2 that almost all of its variables belong to these two categories, most of the program variables fall out of them. Moreover, since there are some small variables in direct $CFSV$ which are also dependent on much bigger variables, indirect $CFSV[0]$ is remarkably larger than direct $CFSV$ in almost all of these programs. As our experiments in live attacks show, the probability of being target data for CFI via DFI attacks in variables in direct $CFSV$ is simpler and thus likely than variables in indirect $CFSV[0]$, then indirect $CFSV[1]$, and so on, respectively. Therefore, by considering the enforcement approach just for variables in direct $CFSV$ and even variables in indirect $CFSV[0]$, as an optimization, the execution and memory overhead decrease. Moreover, by involving critical direct $CFSV$ and their indirect $CFSV$ in the enforcement approach, the overhead is quite negligible.

Fig. 10 depicts the space overhead of the SPEC 2000 benchmarks after applying the CFI via DFI enforcement relative to their execution without instrumentation. In this figure, the available memory overhead of the related work is depicted, and again with N/A columns for programs that some of the related work does not show their overhead. This figure shows the space overhead of CFI via DFI is lower than that of related approaches, e.g. space overhead of DFI is around 150% for all these programs. CFI and shepherding does not impose remarkable memory overhead because of not considering control-data integrity, and thus are not included in this figure, and WIT which just stores colors, and does not offer data recovery, imposes lower data overhead. Similar discussion in the related works shows that the comparison of their overheads in this figure is reasonable.

We also used programs in the SPEC 2006 benchmark suite to measure the runtime and space overhead incurred by our approach which is shown in Fig. 11a and 11b, respectively. As depicted in these figures, the overhead in SPEC 2006 does not exceed those of SPEC 2000. Moreover, although the number of instructions in SPEC 2006 workloads is very larger than the SPEC 2000, since they are mostly among instructions that do not manipulate memory [23], they do not remarkably affect the overhead of our approach. Further, since the SPEC benchmarks focus on CPU intensive programs with integer arithmetics, we believe that the CFI via DFI enforcement causes lower overhead for programs that are not CPU intensive like those that are I/O intensive or those that spend more time in the kernel. Moreover, in some benchmarks such as gcc, memory varies along time instead of growing quickly and stays there. We believe that by proposing an optimization based on our approach by releasing and reusing the memory, the maximum memory overhead would decrease significantly. We work on this optimization as future work.

To evaluate the effectiveness against live exploits, we considered `SSH`, `WU-FTPD`, `NULL-HTTP` and `Telnetd` which violate the CFI via DFI policy. Widely used `SSH` with an overflow overwrites an authentication variable as a decision-making data [4] which is then used to chose the ongoing path at runtime is considered. Also, the most widely used FTP server `WU-FTPD` [11], a data which because of being the user identity data is exploitable to make a CFI via DFI attack successful. Moreover, `NULL-HTTP` server can be attacked via a heap overflow of a configuration data which results in an outbound CFI via DFI violation [6]. In addition, `Telnetd` has a heap buffer overflow of a data which is used for configuration [12]. We examined our approach to prevent these violations against CFI via DFI. The examination result is that the enforcement can prevent all these attacks and recover data after a violation which is not possible in existing approaches. Although it seems that `Portmap`, `Sendmail` [12], and `Cfengine` [13] are exploitable by CFI via DFI attacks, due to the lack of applications semantic knowledge, we could not check them.

Since each of real exploits contains a few vulnerabilities, to perform enough number of attacks to evaluate the proposed technique, it is needed to develop a test suits for CFI via DFI violations as future work. It is because even considering a limited



(a) Execution overhead

(b) Space overhead

Fig. 11. Overhead of CFI via DFI enforcement on SPEC 2006 benchmarks.

number of vulnerabilities, e.g. buffer overflow corresponding to the reports in well-known products by CVE list 2016, a variety of vulnerable code to CFI via DFI attacks can be generated. Although there are some attacks benchmarks such as RIPE [31], since they do not include non-address-control data attacks which their prevention is one of the main differences between this paper and the other related work, we generate our test case. This test case consists of synthetic exploits to evaluate the effectiveness of the enforcement based on a taxonomy of overflow attacks described in [22]. Section VIII-A briefly describes our generated test suite in which we produced and ran instrumented C codes that were compiled without warnings or errors. As our experiments show our enforcement can prevent all the violations in the test suit. While different methods in dynamic analysis tools that use runtime instrumentation to detect memory violations may be sensitive to different sizes of overflows [22], the presented approach dynamically prevents CFI via DFI violations without such limitations. Furthermore, according to [22] just one of the best static buffer overflow detection tool, PolySpace's, with the highest detection rate impose the cost of dramatically long execution times and high false positive and false negative rate.

To check RIPE coverage, we used an over-approximate type-based static analysis (developed in some works such as CPI [30]) that considers different levels of dependencies between code pointers. Instead of point-to analysis in our approach, using this analysis results in RIPE coverage. Considering the same program benchmarks in SPEC 2006 (as in Fig. 11a), the maximum overhead increase was around 4%, belonging to perlbench and gcc.

A. Synthetic Exploits: Generated Test Cases

To evaluate the effectiveness of the CFI via DFI policy enforcement, we used a taxonomy of overflow attacks described in [22] that is developed regarding some attributes to characterize overflow attacks. Each attribute is a parameter with a defined set of possible values such that each kind of overflow can be described by a combination of values of these attributes. Ideally, the test suite should have at least one instance of each possible buffer overflow that could be described by the taxonomy. But, the combination of some of these attributes makes the test suite impractical [22]. Therefore, corresponding our assumptions, we chose some of those attributes to generate codes with various buffer overflows that may result in CFI via DFI violations. Table III shows selected attributes with a brief description and lists some possible values for each attribute. Using attributes in Table III, a basic test case was built such as the code in Fig. 12. Each test case has a $cfsb$ which uses a variable to select the ongoing program control flow at runtime. Also, according to a value for each attribute a buffer overflow is generated in all possible $cfsb.DefSeq, ..., cfsb$ sub-paths of the program.

According to the selected values for attributes in Table III, the buffer overflow in the presented code in Fig. 12, occurred in line 4 which is beyond the upper bound of a stack-based character buffer that is defined and overflowed within the same function. The buffer is not laid within another container and is indexed with a constant. No library function is used for accessing the buffer, the overflow is not within any conditional or complicated control flows and does not depend on the runtime environment. The overflow writes to a discrete location one byte beyond the buffer boundary, and finally, it does not involve a signed vs. unsigned type mismatch. Varying attribute values one at a time, all other test cases are generated.

IX. Conclusions

Attacks that subvert runtime control flow constitute diversity of the most critical and common software attacks. Since these attacks often violate DFI as a prior step to arbitrary control program behavior in real and periodically published exploits, we introduced and enforced a policy called CFI via DFI whose satisfaction can prevent such attacks by imminent detection and recovery of control-data attacks. To the best of our knowledge, there was neither static nor dynamic technique to enforce this policy. More importantly, there was no recovery technique to prevent such attacks in previous work.

```
1: int main(int argc, char* argv[])
2: {int auth=0;
3: char buf[10];
4: buf[10]='A';
4: if (auth==0) then
    6: return 0;
else
    7: return 1;
end
}
```

Fig. 12. A test case of synthetic exploits.

We showed effectiveness of the approach in existing benchmarks and by generating some synthetic exploits. Moreover, the enforcement approach guarantees satisfaction of the policy even in the presence of a powerful adversary with the knowledge of a target program semantics. The approach is applicable to existing programs and comes with no false positive, less false negative and overhead than the most related work with reasonable overhead especially in considering situations in real attacks. Finally, we believe that it is possible to specify this policy as a built-in security feature in a language-based approach, and propose optimization in its enforcement for the variety of its applications as future work.

REFERENCES

[1] J. D. U. Alfred V. Aho, Monica S. Lam, Ravi Sethi, *Compilers: Principles, Techniques, & Tools (Second Edition)*, Addison-Wesley, 2006.
[2] M. Abadi, M. Budiu, et al., "Control-flow integrity principles, implementations, and applications," *ACM Trans. on Inf. and Sys. Sec.*, vol. 13, no. 1, pp. 1-40, Oct. 2009.
[3] J. R. Crandall and F. T. Chong, "Minos: Control Data Attack Prevention Orthogonal to Memory Model," *in 37th Int. Symp. on Micro arch.*, 2004.
[4] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," *in OSDI'06 Proc. of the 7th USENIX Symp. on Op. Sys. Des. and Imp.*, Vol. 7, 2006.
[5] National vulnerability database statistics, *National Institute of Standards and Technology*, http://nvd.nist.gov/statistics.cfm.
[6] S. Chen, J. Xu, et al., "Non-Control-Data Attacks are Realistic Threats," *in Proc. of the 14th USENIX Sec. Symp.*, 2005.
[7] P. Akritidis, C. Cadar, et al., "Preventing Memory Error Exploits with WIT," *IEEE Symp. on Sec. and Priv.*, pp. 263-277, May 2008.
[8] U. Erlingsson, "The Inline Reference Monitors Approach to Security Policy Enforcement," Doctoral thesis, Cornell University, 2004.
[9] S.M. Christey, C.O. Harris et al., "Common weakness enumeration (CWE version 2.5)," *ACM SIGAda Letters*, 2013.
[10] V. Kiriansky, D. Bruening and S. Amarasinghe, " Secure Execution Via Program Shepherding", *in Proc. of the 11th USENIX Sec. Symp.*, 2002.
[11] tf8, "Wu-Ftpd Remote Format String Stack Overwrite Vulnerability," http://www.securityfocus.com/bid/1387,200.
[12] T.-P. Zhang, "RADAR: compiler and architecture supported intrusion prevention, detection, analysis and recovery," Doctoral thesis, Georgia Institute of Technology, Atlanta, 2006.
[13] Pekka Savola, "Very probable remote root vulnerability in cfengine," http://www.shmoo.com/mail/bugtraq /oct00/msg00010.shtml.
[14] T. Ramezanifarkhani and M. Razzazi, "Principles of Data Flow Integrity: Specification and Enforcement," *Inf. Sci. and Eng.*, vol. 31, 2015.
[15] Standard Performance Evaluation Corporation. SPEC CPU 2000, SPEC CPU 2006 benchmark suite, http://www.spec.org.
[16] J. Ferrante, K. J. Ottenstein, and J. D. Warren. "The program dependence graph and its use in optimization," *ACM Trans. on Prog. Lang. and Sys.*, 1987, vol. 9, 3, 319-349.
[17] R. Cytron, J. Ferrante, et al., "An efficient method of computing static single assignment form," *in Proc. of 16th ACM SIGPLAN-SIGACT*, 1989.
[18] L. Andersen. Program analysis and specialization for the C programming language. PhD thesis, University of Copenhagen, 1994.
[19] T. Ramezanifarkhani, M. Razzazi, "Control Flow Integrity via Data Flow Integrity Specification and Dynamic Enforcement," Technical report, Department of Informatics, University of Oslo, Norway, An extended version of this paper, 2018, http://heim.ifi.uio.no/Toktam/Papers/CFIviaDFI.pdf.
[20] C. Zhang, T. Wei, et al., "Practical Control Flow Integrity and Randomization for Binary Executables," in IEEE Symp. on Sec. and Priv., pp. 559-573, 2013.
[21] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," *In Proc. of the 22nd USENIX conf. on Sec.*, 337-352, 2013.
[22] K. R. L. Kratkiewicz, "Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools," *In Proc. BUGS'05*, 2013.
[23] A. Jaleel, "Memory characterization of workloads using instrumentation driven simulation," http://www.glue.umd.edu/ajaleel/workload/, 2010.
[24] D. Jackson, *Software Abstractions*, MIT Press, 2006.
[25] L Shar, L. C. Briand, et al., "Web Application Vulnerability Prediction Using Hybrid Program Analysis and Machine Learning,". IEEE Trans. Dep. Sec. Comput. 12, 6, 688-707, 2015.
[26] E. GÃűktas, E. Athanasopoulos, et al., "Out of Control: Overcoming Control-Flow Integrity," *In IEEE Symp. on Sec. and Pri.*, 575-589, 2014.
[27] N. Carlini, A. Barresi, et al., "Control-flow bending: on the effectiveness of control-flow integrity," In Proc. of the 24th USENIX Conf. on Sec. Symp., 161-176, 2015.
[28] M. Conti, S. Crane, et al., "Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks," *In Proc. of the 22nd ACM SIGSAC Conf. (CCS '15)*, 2015.
[29] MICROSOFT. Phoenix compiler framework. http://research.microsoft.com/phoenix/ phoenixrdk.aspx.
[30] V. Kuznetsov, L. Szekeres, et al., "Code-Pointer Integrity,". *11th USENIX Symp. (OSDI 14)*, 147-163, 2014.
[31] J. Wilander, N. Nikiforakis, et al., "RIPE: runtime intrusion prevention evaluator,". *In Proc. of the 27th (ACSAC '11)*, ACM, 41-50, 2011.
[32] B. Niu, G. Tan, "Modular control-flow integrity," *In Proc. of the 35th ACM SIGPLAN (PLDI '14)*. ACM, 577-587.
[33] Dynamic Linking in Linux and Windows, part one(two), https://www.symantec.com/connect/articles/dynamic-linking-linux-and-windows-part-one(two).