DITTANY: Strength-Based Dynamic Information Flow Analysis Tool for x86 Binaries

Walid J. Ghandour[§]
Univ. Lille, CNRS, Inria
Lille, France
ghandour@utexas.edu

Clémentine Maurice
Univ. Lille, CNRS, Inria
Lille, France
clementine.maurice@inria.fr

***Abstract*—Dynamic dependence analysis monitors information flow between instructions in a program at runtime. Strength-based dynamic dependence analysis quantifies the strength of each dependence chain by a measure computed based on the values induced at the source and target of the chain. To the best of our knowledge, there is currently no tool available that implements strength-based dynamic information flow analysis for x86.**

This paper presents DITTANY, tool support for strength-based dynamic dependence analysis and experimental evidence of its effectiveness on the x86 platform. It involves two main components: 1) a Pin-based profiler that identifies dynamic dependences in a binary executable and records the associated values induced at their sources and targets, and 2) an analysis tool that computes the strengths of the identified dependences using information theoretic and statistical metrics applied on their associated values. We also study the relation between dynamic dependences and measurable information flow, and the usage of zero strength flows to enhance performance.

DITTANY is a building block that can be used in different contexts. We show its usage in data value and indirect branch predictions. Future work will use it in countermeasures against transient execution attacks and in the context of approximate computing.

Keywords*—*binary analysis, dynamic information flow analysis, dynamic slicing, information flow strength, data value prediction, indirect branch prediction.

I. INTRODUCTION

Information flow analysis is a technique that analyzes the flow of information between program objects during execution. It has a wide variety of applications that range from making systems more secure by tracking the dissemination of data inside a system and ensuring some security properties, to helping developers and analysts better understand the code that systems are executing. There exist two types of information flow: static [11], [13], [39] and dynamic [16]. We hereby consider dynamic information flow analysis.

Information flow between objects in a program is crucial to its security since it might reveal tampering or leakage of sensitive information. Security policies can specify the set of objects between which information can flow in a program. Information flow is also important in testing, debugging and maintenance of programs.

In this paper, we present DITTANY, a strength-based dynamic information flow analysis (DIFA) tool that instruments x86 binaries and does not require the availability of the source code. The tool is intended to be generic in a manner to allow several techniques to make use of it. By default, the tool obtains the dependences between all instructions; however, the user can selectively specify which instructions to consider based on their types, instruction pointer (IP), and/or the routines that include them. The tool provides several other options for the user such as considering data dependence only or both data and control dependences. The tool also computes the strengths of the identified dependences. It quantifies information flow in a non-binary manner, and distinguishes between linear and non-linear association, which enables its usage in non-traditional applications such as approximate computing. To the best of our knowledge, currently there is no tool that accomplishes the previously mentioned roles.

DITTANY is a building block that can be used in different contexts. We show its usage in data value and indirect branch predictions. We will use it in future work in countermeasures against transient execution attacks and in the context of approximate computing.

In this paper, we make the following contributions:

- We build a Pin-based dynamic information flow analysis tool that identifies dynamic dependences in a binary executable and records the associated values induced at their sources and targets (Sections IV and VI-A).
- We present an analysis tool that computes the strengths of the identified dependences using information theoretic and statistical metrics applied on their associated values (Sections V and VI-B).
- We use normalized mutual information to identify maximum flow of information from source to target (Section V-A3), show that the existence of data dependence from a source to a target instruction is not a sufficient condition for the flow of information from the source to the target (Sections VII-A and VII-B), and shed light on

[§]The author accomplished this work while he was affiliated with the Lebanese University as continuation to the work that he did during his PhD [19].

Workshop on Binary Analysis Research (BAR) 2022
24 April 2022, San Diego, CA, USA
ISBN 1-891562-76-2
https://dx.doi.org/10.14722/bar.2022.23xxx
www.ndss-symposium.org

exploiting paths that have zero strength flow to increase the effective instruction level parallelism (Section VII-C).
- We introduce dynamic information flow analysis as a tool to study the nature of data and control correlations in programs and to improve the accuracy of instructions outcomes prediction. We present the potential usage of the proposed tool for data value and indirect branch predictions (Section VIII).

II. Background

A. Information Flow

Information flows from a source object x to a target object y whenever information stored in x is passed directly or indirectly to object y [13]. A given program statement designates a specific flow f if executing the statement could result in the flow f. Flows can be either direct or indirect. Direct information flow is due to the propagation of information as a result of data dependence. Source variable is directly involved in the computation of the target variable value. The occurrence of a direct flow is independent of the value of its source object. Assignment, I/O statements, and value-returning procedure calls are examples of statements that result in direct flow. Indirect information flow is due to control dependence. Source object indirectly affects the value of the target object. A statement t is data-dependent on a statement s if t uses a variable that is defined in s. A statement t is control-dependent on a statement s if s decides via branches it controls whether t is executed or not [17], [33], [34].

B. Program Slicing

A *program slice* is the set of program statements that may affect the values at some point of interest. A *static slice* is a set of program statements which *may* affect the value assigned to a variable v at a specific statement S. A *dynamic slice* is a set of program statements that *actually* affect the value assigned to a variable v at a specific statement S for a particular execution of the program. We define an *assembly dynamic slice* (ADS) as the set of assembly instructions that actually affect the value assigned to a register or memory location at a specific instruction *ins* for a particular execution of the program.

C. Indirect Branch Prediction

An indirect branch specifies where the address of the next instruction to execute is located, rather than specifying the address itself, as in a direct branch. Indirect branches can depend on the value of a register or memory location. The target address is not known until the instruction is executed. A single indirect branch can have multiple targets.

Indirect branches implement common programming constructs such as switch statements, virtual function calls and calls through function pointers. Correctly predicting the target addresses of indirect branches can significantly enhance the performance of programs that make a lot of use of the programming features that result in indirect branches.

Most existing indirect branch prediction techniques can be classified as either history-based [9], [14], [15], [23], [29], [24] or precomputation-based [37].

History-based techniques predict the target of an indirect instruction based on its previous history and/or history of previously executed indirect branch instructions. The branch target buffer (BTB) is used to predict the target of the branch as the last taken target [28]. This works well for branches that usually jump to the same target, which are often known as monomorphic indirect branches. However in the case of indirect branches which jump to many different targets, known as polymorphic indirect branches, the prediction accuracy is usually poor.

In this paper, we present a simple profiling-based correlation indirect branch prediction technique that employs dynamic information flow analysis.

D. Value Prediction

Value prediction [30], [32], [48] is a technique to increase parallelism by attempting to overcome serialization constraints caused by true data dependences. By predicting the outcome of an instruction before it executes, value prediction allows data dependent instructions to issue and execute speculatively, hence increasing parallelism when the prediction is correct. In case of a misprediction, the execution is redone with the corrected value. If the benefit from increased parallelism outweighs the misprediction recovery penalty, overall performance could be improved. Enhancing performance with value prediction therefore requires highly accurate prediction methods. Most existing general value prediction techniques are local, i.e., future outputs of an instruction are predicted based on outputs from previous executions of the same instruction. Local value prediction methods can be either computational or context based [48]. A computational predictor yields a predicted next value by performing an operation on previous values such as last value (LV) predictor [32] and stride value (STR) predictor [49]. A 2-δ stride predictor keeps track of two stride values (s_1 and s_2) [35]. A context based predictor makes value prediction based on its observation of previous patterns. An order k Finite Context Method (FCM) predictor uses k preceding values [40]. *Differential Finite Context Method* (DFCM) hashes recent history of strides rather than values and looks up a stride in the hashtable to make a prediction. DFCM has the potential to be more space efficient and faster to warm up than FCM [6].

III. Related Work

Lampson initiated the research on information flow analysis and listed a number of possible information leaks [27]. Fenton introduced an abstract machine, named Data Mark Machine, that supports dynamic checking of information flows [16]. Jones and Lipton presented a similar mechanism [22]. Zimmerman et al. showed an intrusion detection model that employs runtime enforcement of an information flow policy that dictates which information flows are allowable in a given system [53], [52]. Their model focus on information flow between entire objects. Haldar et al. extended the Java Virtual Machine (JVM) to enforce information flow policies at the granularity of objects without considering control dependences [21].

Static information flow analysis techniques are less precise than dynamic techniques since they consider all control flows as executable [13].

Algorithm 1: Data dependence analysis

1. Get the sets of dependences of the source operands;
2. Unify the sets obtained in step 1;
3. Unify the set obtained in step 2 with the address of the executed instruction;
4. Clear the dependence sets of the target operands and assign to it the set obtained in step 3;
5. Set the dependence set of the executed instruction to its union with the set obtained in step 2;

Algorithm 2: Control dependence analysis

1. When a conditional branch instruction bc is executed, its set of dependences depend is computed, and bc is pushed on $controlDepStack$;
2. When the immediate post-dominator $ipdom$ of bc is executed, bc is popped off $controlDepStack$;
3. When an instruction ins is executed, Top of Stack (TOS) and its dependences are added to the sets of dependences of ins and its target operands;
4. Pop TOS off $controlDepStack$ when a branch instruction with the same $ipdom$ is reached. This ensures that the stack size is bounded;

Program slicing was originally introduced by Weiser [47] who proposed the first static slicing algorithm to help debugging programs. Afterward, Korel and Laski [25] proposed dynamic slicing that considers runtime information when selecting the set of statements that affects a specific program variable. The previously mentioned algorithms use *backward* analysis, which requires the a priori availability of the execution trace. Korel and Yalamanchili were the first to propose a forward-computing algorithm for dynamic slicing [26]. Tip [42] presented a survey of both static and dynamic program slicing techniques. Program slicing has been explored for Web applications [31], [43], [36] and for Android [51], [50], [5], [2].

DynFlow is a tool that supports strength-based dependence analysis of Java programs [45]. It calculates flow strength at the bytecode level whereas we are interested in obtaining the results the tool provides at the assembly level for C and C++ programs. Forward computing dynamic slicing and information flow analysis algorithms, which we use in this paper, do not need the a priori availability of the execution trace.

IV. Dynamic Dependence Implementation

In this section, we describe our approach to dynamic dependence analysis.

A. Data Dependence Analysis

We define *instruction target operands* as the instruction destination operand and any implicit operand whose value might change when the instruction executes and *instruction source operands* as the explicit and/or implicit source operands of the instruction. We also define a mapping function to each instruction opcode. After the dynamic execution of each assembly instruction, we calculate the assembly dynamic slice of the target operands, by using instruction-type related mapping function that takes as inputs the assembly dynamic slices of the source operands of the instruction. The instruction itself is added to the dynamic slices of its target operands. The new set of dependences of the executed instruction is obtained as the union of its existing set with those of its source operands. Algorithm 1 summarizes the steps.

We mean by *unify* the union operation of two or more sets. In set theory, the union of a collection of sets is the set of all elements in the collection. According to the opcode of the executed instruction, we identify the set of source and the set of target operands. We obtain the set of dependences of each of the source operands and unify the obtained set of dependences. We name the obtained set *SourceDepSet*. We unify *SourceDepSet* with the address of the executed instruction and assign the result to a set that we name *DepSet*. We clear the dependence sets of the target operands and assign to it *DepSet*. We set the dependence set of the executed instruction to its union with *SourceDepSet*.

B. Control Dependence Analysis

A *Control Flow Graph (CFG)* is a directed graph augmented with two specials nodes *entry* and *exit* with no predecessors and no successors, respectively. A node of the graph corresponds to a basic block, i.e., a single-entry single-exit sequence of instructions, while an edge represents the potential flow of control from one basic block to another, branches or fall-through execution. A CFG node n post-dominates another node m (*n pdom m* or *pdom(m) = n*) if every path from m to *exit* goes through n. Node n is the immediate post-dominator of node m (*n ipdom m* or *ipdom(m) = n*) iff $m \neq n$; *n pdom m*; $\forall d \neq m$, *d pdom m* $\rightarrow$ *d pdom n*. Every node has a unique immediate post-dominator.

A node n of a CFG is control-dependent on a node c if (1) there exists an edge e_1 coming out of c that definitely causes n to execute, and (2) there exists some edge e_2 coming out of c that is the start of some path that avoids the execution of n. The decision made at c affects whether n gets executed. Formally, n is control-dependent on c if: (1) $n \neq c$, (2) n does not post-dominate c, and (3) there exists a path from c to n such that n post-dominates every node in the path except c [3], [4], [17].

To track control flow-based dependences, we use an approach similar to the one originated in [44], [45]. The outcome of a conditional branch dictates whether the instructions that depend on it are executed or not. The values of operands that affect the branch outcome may affect the dependent instructions target operands values. Thus, we need to add the set of dependences of the source operands of the branch instruction to the set of dependences of the targets operands of the branch dependent instructions. To achieve this, we leverage dynamic slicing and static post-dominance information using Algorithm 2.

The algorithm uses a stack, denoted *controlDepStack*, which holds pointers to branch instructions whose *dynamic control scope* is still active. The *dynamic control scope* of a branch instruction terminates when its *ipdom* is executed, at

which time, the branch instruction is popped off *controlDepStack*. To ensure that the stack size is bounded, when a newly encountered branch instruction has the same *ipdom* as that of the branch instruction at the top of the stack *controlDepStack*, denoted *TOS*, we pop *TOS* off *controlDepStack*. This is acceptable as explained in [45].

C. Recording Values

We determine the assembly dynamic slices for selected or all executed instructions in a program. We are interested in quantifying the amount of flow from a source instruction to a target instruction. To achieve this for non-branch source instructions, we first need to log pairs of values corresponding to consecutive dynamic executions of both instructions. When the source of a flow is a branch, we store the status of the branch, taken or non-taken, for each occurrence of the flow.

We use a map structure to associate with each instruction the outcome of its last execution. We create a `ValuePairs` class that holds pair of values corresponding to consecutive dynamic execution of source and target instructions. We create another class, `Depend`, that is instantiated for each pair of dependent instructions – dependence does not have to be strictly between distinct instructions, it can be from the instruction to itself. After the dynamic execution of each instruction, we update its corresponding `Depend` instance with pairs of values of its last outcome and the last outcomes of instructions on which it depends during its current execution. In fact, the dependences of an instruction may change from an execution to another.

V. Dependence Strength

Informally, information flow occurs from an instruction ins_1 to an instruction ins_2 during execution of a program if observing the outcome of ins_2 at some point reduces one's uncertainty about the outcome of ins_1 at an earlier point, which indicates that ins_2 is probabilistically dependent on ins_1. The strength of dependences measures the amount of information they propagate. We follow an approach similar to [46] to calculate the strength of dependences.

We use three techniques to measure information flow strength. The first employs entropy, conditional entropy and mutual information. The other two are correlation-based; one employs *standard r* and the other, *eta coefficient*. *Standard r* measures linear relations only; however, information theoretic and *eta coefficient* measure both linear and non-linear relations. Therefore, we can identify and separate linear and non-linear correlations.

A. Entropy-Based Technique

1) Entropy: Entropy is a measure of the uncertainty of a random variable. It explicitly quantifies the information content in a given source of data. The entropy of a discrete random variable X is defined as [12]:

$$H(X) = -\sum_i P(X = x_i) \log_2 P(X = x_i).$$

Conditional entropy of a random variable X, given that the random variable Y is known, measures the remaining uncertainty about the value of X after knowing the value of Y. It is defined as:

$$H(X|Y) = -\sum_j P(Y = y_j) \times \sum_i P(X = x_i|Y = y_j) \log_2 P(X = x_i|Y = y_j).$$

$H(X|Y) = 0$ if and only if the value of X is completely determined by the value of Y. $H(X|Y) = H(X)$ if and only if X and Y are independent random variables.

2) Mutual Information: Mutual information is a measure of the amount of information one random variable contains about another. The mutual information between two discrete random variables X and Y is given by:

$$\begin{aligned} I(X;Y) &= H(X) - H(X|Y) \\ &= H(Y) - H(Y|X) \\ &= H(x) + H(Y) - H(X,Y), \end{aligned}$$

where $H(X,Y)$ is the joint entropy. Mutual information is always non-negative. The mutual information $I(X,Y)$ is the reduction in the uncertainty of X due to the knowledge of Y. Mutual information is symmetric, $I(X;Y) = I(Y;X)$. Thus X says as much about Y as Y says about X.

Mutual information can detect arbitrary *non-linear* relationships between X and Y. $I(X;Y)$ can be normalized to a value between 0 and 1:

$$NMI(X;Y) = \frac{H(X) - H(X|Y)}{H(X)} = \frac{H(Y) - H(Y|X)}{H(Y)}$$

$$NMI(X;Y) = \begin{cases} 1 & \text{for } H(X|Y) = 0, \\ 0 & \text{for } H(X|Y) = H(X). \end{cases}$$

3) Entropy-Based Strength of Flow: Following the analysis presented in [46], the amount of information transferred from x (in state σ) to y (in state τ) is:

$$StrengthFlow_{Entropy}(x,y) = H(X_\sigma) - H(X_\sigma|Y_\tau)$$

$StrengthFlow_{Entropy}(x,y)$ will be zero if no measurable flow occurred between x and y, and it will be greater than zero otherwise. The value that results from this equation does not have a range limit.

We employ normalized mutual information as a technique to measure the strength of flow, since it results in a value between 0 and 1, where 0 indicates no flow of information, and 1 indicates maximum flow of information. Maximum flow of information implies that if the value of the source variable is known, we can definitely deduce the value of the target variable (no uncertainty remains about the value of the target variable knowing the value of the source variable).

$$StrengthFlow_{EntropyDITTANY}(x,y) = \frac{H(X_\sigma) - H(X_\sigma|Y_\tau)}{H(X_\sigma)}$$

Normalized mutual information allows us to identify cases where there is maximum flow of information from source

to target. Previous work [46] only reasons about weak and strong flow, with no additional interpretation of the values obtained for measured strengths of flows. Moreover, it does not provide a criteria to classify a strength of flow (either weak or strong), and does not address the issue of maximum flow of information.

B. Correlation-Based Techniques

We now provide some information about correlation-based strength of flow techniques we use in our tool.

1) Standard r: The Pearson product moment correlation coefficient, which is also known as *Pearson's r* or *standard r*, measures the strength of the correlation (linear dependence) between two variables. It ranges between -1 and $+1$ and is defined as [38]:

$$r = \frac{Cov(x, y)}{\sigma_x \sigma_y}$$

where $Cov(x, y)$ is the covariance of x and y, σ_x is the standard deviation of x, and σ_y is the standard deviation of y. r^2 is called the coefficient of determination or proportion of explained variance.

When two variables have a nonlinear relationship, standard r will underestimate the strength of the relationship. $StrengthFlow_r(x, y)$ is the absolute value of *standard r*.

2) Eta Coefficient: The correlation ratio or *eta coefficient* provides the same value as *standard r* for linearly related variables and a greater value for non-linearly related variables. The difference between *eta* and *r* presents a measure of the non-linearity of the relation between two variables. The *eta coefficient* is defined as:

$$eta = \frac{\sigma_{\overline{y}}}{\sigma_y}$$

where $\overline{y}$ is the mean of the category that y belongs to, $\sigma_{\overline{y}}$ is the standard deviation of $\overline{y}$, and σ_y is the standard deviation of y [41], [10].

In the case that either x or y is constant, $StrengthFlow_r(x, y)$ is defined to be zero, because there is no linear association between x and y. Similarly, in the case that y is constant, $StrengthFlow_{eta}(x, y)$ is defined to be zero, because there is no linear or nonlinear association between the variables.

VI. DITTANY

We implement our approach to dynamic dependence analysis in a tool that we call DITTANY[1]. The tool leverages Pin dynamic instrumentation tool.

A. Information Flow Analysis

Figure 1 presents an overview of DITTANY mode of operation for dynamic information flow analysis. By default, the tool tracks flow of information among all dynamically executed

[1]Dittany can refer to three different plants: White Dittany (Dictamnus Albus), Dittany of Crete (Origanum Dictamnus), and Common Dittany (Cunila Mariana). DITTANY: Dynamic Information and TainTing flow ANalYsis tool. We have implemented the strength based information flow analysis phase, and we still have the tainting part.

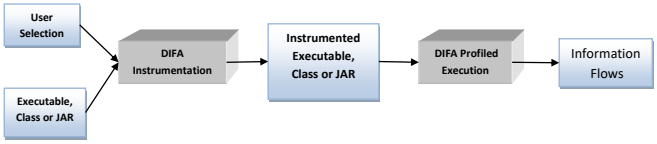

Fig. 1. Overview of DIFA tool operation.

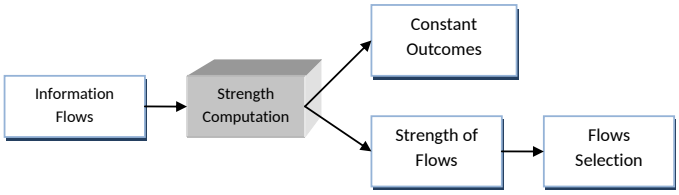

Fig. 2. Overview of strength of flow computation.

instructions. We provide several command line options that enable the user to obtain flow of information along data dependences only or along both data and control dependences, select a particular routine to instrument, specify the routine at which instrumentation starts and the routine at which it terminates, indicate the number of instructions to skip before starting instrumentation, choose a particular path of interest, mention an upper limit on the number of instructions to instrument, etc.

When the user specifies the types of instructions to instrument, Step 5 of Algorithm 1 is only executed for the specified instructions. For non-specified instructions, Step 3 of Algorithm 2 only updates their target operands without updating their sets of dependences. Taking into consideration the user selection, the tool instruments the input executable which is run to produce profiles that capture the induced information flows along with their respective source and target values.

B. Strength Calculation

Figure 2 presents an overview of DITTANY mode of operation for the strength of flow computation. We parse the information flows obtained to determine source and target instructions, and calculate the strength of flow from source to target using information theory, *standard r* and *eta* coefficient. The tool also identifies instructions which outcomes are constant during all executions. The flows can be categorized according to their strength using one or more of the used strength measuring schemes.

VII. DYNAMIC DEPENDENCE AND INFORMATION FLOW

We address in this section the relation between dependences and measurable information flow, and the usage of zero strength flows to enhance performance. The work explored in Sections VII-A and VII-B below, was addressed in different context in [46].

A. Dependences and Measurable Information Flow

We answer in this section the following question: *Are dynamic dependences generally indicative of measurable information flow?*

```
1    for (int x=0; x<20; x++){
2      a = x - x%5;
3      y = a + 1;
4    }
```

Listing 1. C++ example 1.

```
1    movl  $0, -12(%rbp)
2    jmp .L2
3    .L3:
4    movl  -12(%rbp), %ecx : (ins1), x value
5    movl  $1717986919, -36(%rbp)
6    movl  -36(%rbp), %eax
7    imull %ecx
8    sarl  %edx
9    movl  %ecx, %eax
10   sarl  $31, %eax
11   movl  %edx, %ebx
12   subl  %eax, %ebx
13   movl  %ebx, -28(%rbp)
14   movl  -28(%rbp), %eax
15   sall  $2, %eax
16   addl  -28(%rbp), %eax
17   movl  %ecx, %edx
18   subl  %eax, %edx
19   movl  %edx, -28(%rbp)
20   movl  -12(%rbp), %eax
21   subl  -28(%rbp), %eax
22   movl  %eax, -24(%rbp) : (ins2), a value
23   movl  -24(%rbp), %eax
24   addl  $1, %eax
25   movl  %eax, -16(%rbp) : (ins3), y value
26   addl  $1, -12(%rbp) : x++
27   .L2:
28   cmpl  $19, -12(%rbp)
29   jle .L3
30   movl  $0, %eax
31   popq  %rbx
32   leave
33   ret
```

Listing 2. x86 assembly of example 1.

Consider Listing 1 and its corresponding assembly code Listing 2. It represents a for loop that contains two statements. The first statement assigns to a variable a the result of the subtraction of x modulus 5 from x. The second statement assigns $a + 1$ to variable y.

The outcomes of the instructions labeled ins_1, ins_2 and ins_3 correspond to the values of x, a, and y in the source code respectively. Both instructions ins_2 and ins_3 are data-dependent on instruction ins_1. Instruction ins_3 is data-dependent on instruction ins_2.

When executing the program, the following (ins_1, ins_2, ins_3) value triplets result:

(0, 0, 1), (1, 0, 1), (2, 0, 1), (3, 0, 1), (4, 0, 1), (5, 5, 6), (6, 5, 6), (7, 5, 6), (8, 5, 6), (9, 5, 6), (10, 10, 11), (11, 10, 11), (12, 10, 11), (13, 10, 11), (14, 10, 11), (15, 15, 16), (16, 15, 16), (17, 15, 16), (18, 15, 16), (19, 15, 16).

Computing the strength of flows (ins_1, ins_2), (ins_1, ins_3) and (ins_2, ins_3) using normalized mutual information yields 0.46, 0.46 and 1.0 respectively. The low value of the strength of flow (ins_1, ins_2) is expected since by learning that the outcome of ins_2 is 0, an observer is not certain of the outcome of ins_1, which could be $0, 1, 2, 3$ or 4. Similarly for the strength of flow (ins_1, ins_3). The strong value of the strength of flow (ins_2, ins_3) is justified by the fact that learning the outcome of ins_3, we can infer the outcome of ins_2 with 100% certainty.

```
1    for (int x=0; x<20; x++){
2      a = x - x*5;
3      y = a + 1;
4    }
```

Listing 3. C++ example 2.

```
1    movl  $0, -4(%rbp)
2    jmp .L2
3    .L3:
4    movl  -4(%rbp), %edx
5    movl  %edx, %eax
6    sall  $2, %eax
7    leal  (%rax,%rdx), %edx
8    movl  -4(%rbp), %eax
9    subl  %edx, %eax
10   movl  %eax, -16(%rbp) : (ins2), a value
11   movl  -16(%rbp), %eax
12   addl  $1, %eax
13   movl  %eax, -8(%rbp) : (ins3), y value
14   addl  $1, -4(%rbp)  : (ins1), x++ value
15   .L2:
16   cmpl  $19, -4(%rbp)
17   jle .L3
18   movl  $0, %eax
19   leave
20   ret
```

Listing 4. x86 assembly of example 2.

Listing 3 is another example with its corresponding assembly code Listing 4. The only difference between this example and the previous one is the usage of multiplication instead of modulus. Computing the strength of flows (ins_1, ins_2), (ins_1, ins_3) and (ins_2, ins_3) using normalized mutual information yields 1.0 for all of them.

We can conclude from the above two examples that *the presence of dynamic dependence between two instructions is not a sufficient condition for the strong flow of information between them.* Other program constructs and constraints are beyond the scope of this paper.

B. Zero Strength Flows Interpretations and Analyses

Zero strength flows are dependences where there are no correlation between the values of the source and those of the target. The strength of a zero strength flow measured using *eta* coefficient, normalized mutual information and standard r equals to 0.

We evaluate our metrics on the SPEC CINT2006, i.e., SPEC CPU 2006 integer benchmarks [1] described in Table III in Appendix. We use all CINT2006 benchmarks except *bzip2* due to a bug.

We define *self flow* as the flow from and instruction into itself. Figure 3 shows that Self Zero Strength r, Self Zero Strength ETA and Self Zero Strength Entropy accounts for

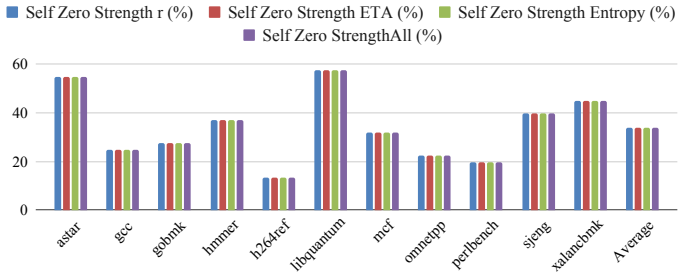

Fig. 3. Percentage of zero strength flows of the total dynamic flows from instructions into themselves.

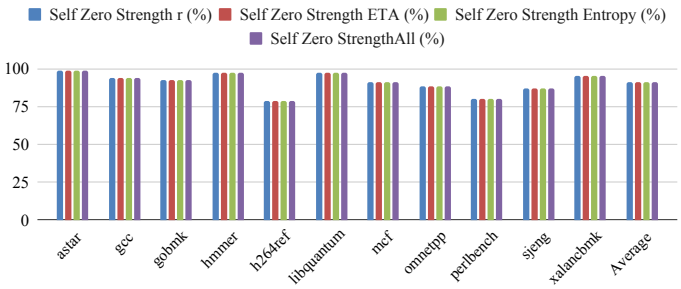

Fig. 4. Percentage of zero strength flows of the total dynamic flows between distinct instructions.

33.877% of the total dynamic flows from instructions into themselves. More than third of the self flows have strength exactly equal to zero.

We define *distinct flow* as the flow from an instruction into another distinct instruction. Figure 4 shows that Distinct Zero Strength r, Distinct Zero Strength ETA and Distinct Zero Strength Entropy accounts for 90.94%, 90.927% and 90.917% of the total dynamic flows between distinct instructions. More than 90% of the distinct flows have strength exactly equal to zero.

It is proven in [33], [34] that a required condition for a change of the outcome computed by statement s_1 to impact the execution behavior of statement s_2 is the existence of static program dependences between s_1 and s_2.

The results presented in this section imply that the existence of dynamic dependences between two statements s_1 and s_2 is not a sufficient condition for the outcome produced by s_1 to influence the outcome produced by s_2.

In addition, these results imply a detected insecure information flow is highly probable to be benign due to the fact that no quantified information is transmitted.

C. Zero Strength Flows and Instruction Level Parallelism

In this section, we answer the following question: *Can we exploit paths that have zero strength flow to increase the effective instruction level parallelism (ILP)?*

To answer this question, we do the following: (1) use DITTANY to identify the apparent information flows that occurred during the execution of several programs, (2) compute the strength of the identified flows, i.e., measure the amount of information they propagated, (3) select zero strength flows between distinct instructions where the source of flows are load instructions, and (4) analyze the predictability of the values of the source and the target of the selected flows.

For SPEC CPU 2006 integer benchmarks [1], we measure the prediction accuracy and coverage of the source and the target of flows using last value [32], 2-δ stride [35] and DFCM [6] predictors. Prediction accuracy is the number of successful predictions over the total number of attempted predictions. Prediction coverage is the number of an instruction predictions over the total number of an instruction updates.

Figure 5 shows that the target instructions are predictable on average with 100%, 95.477% and 97.455% accuracies using last value, 2-δ stride and DFCM predictors respectively. The prediction coverages achieved on average using last value, 2-δ stride and DFCM predictors are 67.2%, 70.53% and 86.77% respectively.

Figure 6 shows that 40.04% of the selected source load instructions are selected for confident predictions, and 86% of the selected target instructions are selected for confident predictions. We use confidence counters to guide when to use the prediction information.

Figure 7 shows that the selected load instructions and target instructions account on average for 0.705% and 3.58% respectively of the total number of dynamically executed instructions.

We can conclude from our results that 86% of the target of zero strength flows, with load instructions as the source of the flows, are predictable with very high accuracy using local prediction techniques, and that they account for around 3.08% of the totally dynamically executed instructions. We also observe that the prediction accuracies of the source and target of the same zero strength flow are not generally correlated. Holding a highly predictable target instruction from execution since it is connected via zero strength flow with a non-predictable load instruction can negatively impact the execution performance. We can attribute the results we obtained to the following:

- The results are related to zero strength flows, i.e. flows that have zero strength using all the three schemes used.
- Neither linear nor non-linear association exists between the source and target values.
- Highly predictable target instructions have strong self linear flow, which results in being highly predictable using local prediction techniques.

Data dependence occurs when an instruction reads a register or a memory location whose values depend on the execution of a previous instruction. Data dependence presents a limiting factor on ILP. We show that the existence of data dependence from a source to a target instruction is not a sufficient condition for the flow of information from the source to the target. Further work is required to check the possibility of exploiting paths that have zero flow strength to increase the effectiveness of ILP. Such paths are dependences where the source variables do not matter in the computation of the target variables values, thus making them irrelevant dependences to the execution.

VIII. APPLICATIONS

In this section we describe the application of DITTANY in data value and indirect branch predictions.

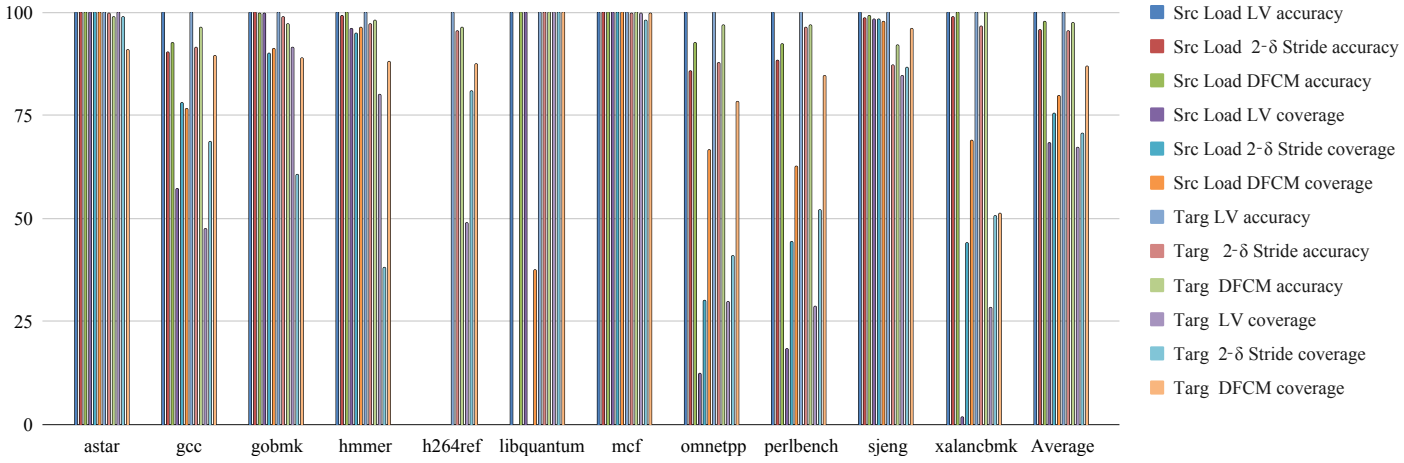

Fig. 5. Prediction accuracy and coverage of selected distinct instructions related via zero strength flows.

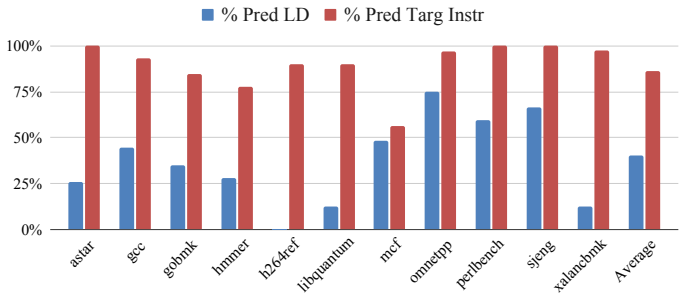

Fig. 6. Distributions of predictability of source and target instructions related via zero strength flows.

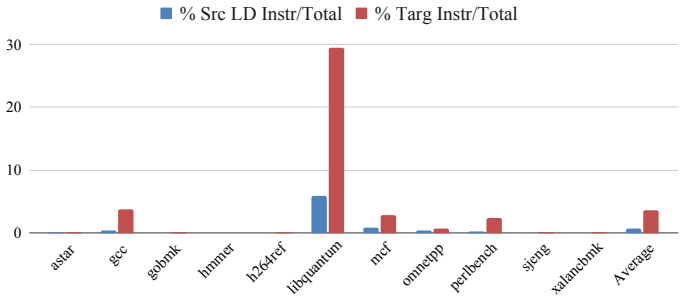

Fig. 7. Selected instructions out of the total dynamically executed instructions.

A. Application to Indirect Branch Prediction

We use DITTANY to identify the last instruction that updates the source operand of an indirect branch. We call this instruction the source instruction of the indirect branch. For an indirect branch instruction that is not highly predictable using the regular BTB technique [28], we predict its target address by accessing the prediction table using a value obtained by hashing the instruction pointer of the predicted indirect branch instruction with the outcome of its source instruction. We refer to this technique as BTBSrc. Our hybrid (BTB, BTBSrc) predictor is a combined predictor that remembers which of the predictors has made the best predictions in the past.

a) Benchmarks Characteristics: Table I presents the characteristics of indirect branch instructions in SPEC CPU 2006 benchmarks. We report the number of static instructions and static indirect branch instructions as well as the percentage of static indirect branch instructions with respect to the total number of static instructions. In addition, we present the percentage of dynamic indirect branch instructions with respect to the total number of dynamically executed instructions. We notice that indirect branch instructions account for 0.16% and 0.089% of the total static and dynamic instructions respectively.

b) Prediction Accuracy: Table II presents the prediction accuracy of indirect branch instructions using BTB and hybrid (BTB, BTBSrc). We use 16K tables for both BTB and BTBSrc. BTB prediction accuracy is 84.28%, while the hybrid (BTB, BTBSrc) achieves 95.46% prediction accuracy on average. BTBSrc alone with unlimited table size achieves 100% accuracy.

B. Application to Data Value Prediction

Using DITTANY, we select all instructions that have flows into them characterized by $|r| = 1$. We categorize the flows as follows: (i) flow from the instruction to itself, or (ii) flow from a source instruction into a target instruction. We define *self linear relation* type as the set of instructions that satisfy criterion (i) with strength of flow measured using *standard r* equals to 1, and *distinct linear relation* type as the set of instructions that satisfy criterion (ii) and $|r| = 1$.

Using DITTANY, instructions are classified as either constant outcome instructions, or variable outcome instructions. Constant outcome instruction is an instruction that always produces the same value. Constant outcome instructions can be predicted with high accuracy using local last value predictor. For variable outcome instructions, we select instructions that exhibit (i) linear flow from the instruction to itself, and (ii) linear flow from a source instruction into the instruction of interest. For the instructions that satisfy criteria (i), we employ several local prediction techniques. For the instructions that satisfy criteria (ii), we predict the value y of the instruction of interest, knowing the value x of the source instruction using the linear equation $y = ax + b$. We use this technique when the instruction of interest neither follow a linear nor a FCM

TABLE I. CHARACTERISTICS OF SPEC CPU 2006 EVALUATED BENCHMARKS.

	astar	gcc	gobmk	hmmer	h264ref	libquantum	mcf	omnetpp	perlbench	sjeng	xalancbmk
Total Static Instr.	12224	848132	234509	89025	166161	10215	3389	152468	298936	32012	962864
Static Indir. Br.	26	571	66	93	59	30	22	122	200	57	237
%Static Indir. / Total Static Instr.	2.13×10^{-1}	6.73×10^{-2}	2.8×10^{-2}	1.044×10^{-1}	3.55×10^{-2}	2.93×10^{-1}	6.5×10^{-1}	8×10^{-2}	6.7×10^{-2}	1.78×10^{-1}	2.46×10^{-2}
%Dynamic Indir. / Total Dynamic Instr.	7.32×10^{-4}	2.3×10^{-3}	4×10^{-3}	6.8×10^{-3}	7.22×10^{-3}	1.9×10^{-3}	2.86×10^{-4}	9.45×10^{-2}	6.4×10^{-1}	2.05×10^{-1}	2.31×10^{-2}

TABLE II. PREDICTION ACCURACY USING BTB AND HYBRID (BTB, BTBSRC).

	astar	gcc	gobmk	hmmer	h264ref	libquantum	mcf	omnetpp	perlbench	sjeng	xalancbmk	average
BTB	90.76%	65%	89.05%	92.93%	91.05%	96.54%	83.33%	90.52%	74.29%	70.56%	83.06%	84.28%
Hybrid(BTB, BTBSrc)	90.76%	87.6%	100%	92.93%	100%	96.54%	83.33%	100%	98.95%	100%	100%	95.46%

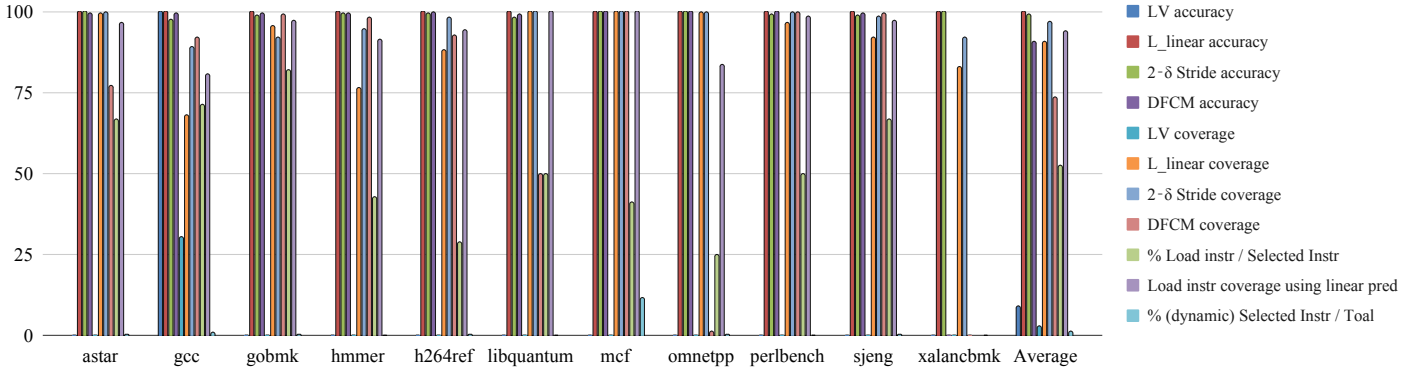

Fig. 8. Self linear relation type instructions measurements.

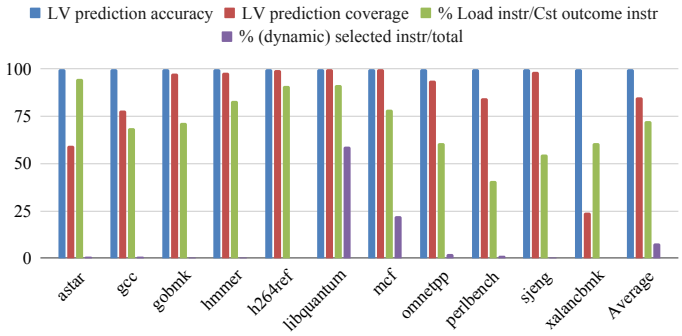

Fig. 9. Constant outcome instructions measurements.

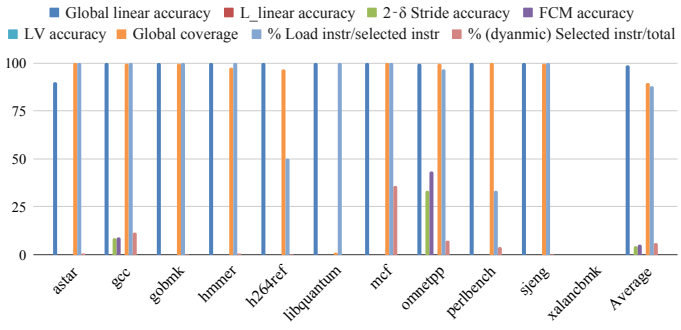

Fig. 10. Distinct linear relation type instructions measurements.

pattern, e.g., it is not predictable using any local prediction technique.

a) Value Prediction Pintool: We develop a pintool that implements last value, local linear, 2-δ stride, DFCM, and global linear predictors. A local linear predictor is a predictor that implements the linear relation, $y = ax + b$, to predict the outcome of an instruction from its previous value [18], [20]. A global linear predictor is similar to a local linear predictor except that it uses the value of a source instruction that has a strong linear information flow into the target instruction to predict the outcome of the target instruction [18], [20].

b) Empirical Results: We now report empirical results obtained from applying DIFA directed value prediction to the CINT2006 benchmarks. We select the instructions to predict using DITTANY and use the pintool described above. Unlimited table size is assumed for the results reported in Figures 8, 9 and 10.

Figure 9 shows that the last value predictor (LV) achieves on average 100% and 84.84% prediction accuracy and prediction coverage respectively for the selected *constant outcomes instructions*. These instructions account for 7.9% on average of the totally dynamically executed value producing instructions. Loads constitute 72.46% on average of these instructions.

Figure 8 shows that the instructions of the *self linear relation* type are highly predictable using local value prediction techniques. The local linear, 2-δ stride and DFCM predictors achieve on average 100%, 99.2%, and 90.56% prediction accuracy, and 90.81%, 96.7% and 73.6% prediction coverage respectively. Loads constitute 52.37% on average of these instructions.

Figure 10 shows that the global linear predictor achieves 98.98% prediction accuracy and 89.47% prediction coverage on average for instructions of the *distinct linear relation* type. It significantly outperforms all local predictors for this set

of instructions. Instructions of this category present 6.1% on average of the totally dynamically executed value producing instructions. Loads constitute 88% on average of these instructions.

We have previously employed DITTANY in our work on value prediction [18].

IX. Future Work

Modern processors employ branch prediction and speculative executions to improve their performance, which have been shown to be prone to transient execution attacks [8]. In future work, we plan to use DITTANY to introduce new countermeasures using the information collected by the tool to prevent miss-training of prediction tables, and to introduce safe prediction techniques that are robust against existing side channels attacks. We will introduce approximate prediction techniques by leveraging our strength-based approach to quantify information flow in a non-binary manner. In addition, we will port DITTANY to DynamoRIO [7] since it supports IA-32, AMD64, ARM, and AArch64. Furthermore, we will implement dynamic information flow analysis in hardware, so that we can integrate our proposed techniques in real designs.

X. Conclusion

We described a Pin-based dynamic information flow analysis tool that identifies dynamic dependences in a binary executable and records the associated values induced at their sources and targets. We also computed the strengths of the identified dependences using information theoretic and statistical metrics applied on their associated values. We showed that the existence of data dependence from a source to a target instruction is not a sufficient condition for the flow of information from the source to the target. We presented and evaluated the usage of the proposed tool in data value and indirect branch predictions using SPEC CPU 2006 integer benchmarks.

Availability. The tool is publicly available with a detailed user guide under GNU GPLv3 at DITTANY repository.

Acknowledgments

This work benefited from the support of the project ANR-19-CE39-0008 ARCHI-SEC.

References

[1] "The Standard Performance Evaluation Corporation (SPEC). The SPEC Benchmark Suite," http://www.spec.org.

[2] K. Ahmed, M. Lis, and J. Rubin, "Mandoline: Dynamic slicing of android applications with trace-based alias analysis," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 105–115.

[3] A. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[4] F. E. Allen, "Control flow analysis," *Proceedings of a symposium on Compiler optimization*, vol. 5, July 1970.

[5] T. Azim, A. Alavi, I. Neamtiu, and R. Gupta, "Dynamic slicing for android," in *ICSE*, 2019.

[6] B. Goeman, H. Vandierendonck and K. De Bosschere, "Differential fcm: Increasing value prediction accuracy by improving table usage efficiency," *HPCA'01*.

[7] D. Bruening, T. Garnett, , and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *CGO*, 2003.

[8] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *USENIX Security Symposium*, 2019.

[9] P. Chang, E. Hao, and Y. N. Patt, "Target prediction for indirect jumps," *ISCA'97*.

[10] D. J. Krus, "Visual statistics: Chapter 14, regression on multiple categories, visual-statistics.net," 2006.

[11] D. E. Denning, "A lattice model of secure information flow," *Comm. ACM*, vol. 19, no. 5, pp. 236–242, 1976.

[12] ——, *Cryptography and Data Security*. Addison-Wesley, 1982.

[13] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communication of the ACM*, vol. 20(7), pp. 504–513, 1977.

[14] K. Driesen and U. Holzle, "Accurate indirect branch prediction," *SIGARCH Comput. Archit. News*, vol. 26, no. 3, 1998.

[15] ——, "The cascaded predictor: economical and adaptive branch target prediction," *MICRO 31*, 1998.

[16] J. S. Fenton, "Memoryless subsystems," *The Computer Journal*, vol. 17, no. 2, pp. 143–147, 1974.

[17] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, July 1987.

[18] W. J. Ghandour, H. Akkary, and W. Masri, "Leveraging strength-based dynamic information flow analysis to enhance data value prediction," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 1, pp. 1–33, Mar. 2012.

[19] W. J. Ghandour, "Leveraging strength-based dynamic information flow analysis to enhance control and data value predictions," Ph.D. dissertation, The American University of Beirut, 2011.

[20] W. J. Ghandour, H. Akkary, and W. Masri, "The potential of using dynamic information flow analysis in data value prediction," *PACT '10*, pp. 431–442, September 2010.

[21] V. Haldar, D. Chandra, and M. Franz, "Practical, dynamic information flow for virtual machines," in *2nd International Workshop on Programming Language Interference and Dependence*, 2005.

[22] A. K. Jones and R. J. Lipton, "The enforcement of security policies for computation," *Journal of Computer and System Sciences*, vol. 17, no. 1, pp. 35–55, 1978.

[23] J. Kalamatianos and D. R. Kaeli, "Predicting indirect branches via data compression," *MICRO-31*, 1998.

[24] H. Kim, J. A. Joao, O. Mutlu, C. J. Lee, Y. N. Patt, and R. Cohn, "Vpc prediction: Reducing the cost of indirect branches via hardware-based dynamic devirtualization," *ISCA-34*, 2007.

[25] B. Korel and J. Laski, "Dynamic program slicing," *Inf. Process. Lett.*, vol. 29, pp. 155–163, October 1988.

[26] B. Korel and S. Yalamanchili, "Forward computation of dynamic program slices," in *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, 1994, pp. 66–79.

[27] B. W. Lampson, "A note on the confinement problem," *Communications of the ACM*, vol. 16, no. 10, pp. 613–615, 1973.

[28] J. K. F. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer*, vol. 17, no. 1, 1984.

[29] T. Li, R. Bhargava, and L. K. John, "Adapting branch-target buffer to improve the target predictability of java code," *ACM Trans. Archit. Code Optim.*, vol. 2, no. 2, pp. 109–130, June 2005.

[30] M. H. Lipasti and J. P. Shen, "Exceeding the dataflow limit via value prediction," in *MICRO-29*, 1996.

[31] J. Maras, J. Carlson, and I. Crnković, "Client-side web application slicing," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 504–507.

[32] M.H. Lipasti, C.B. Wilkerson and J.P. Shen, "Value locality and load value prediction," in *ASPLOS*, 1996.

[33] A. Podgurski, "Significance of program dependences for software testing, debugging and maintenance," Ph.D. dissertation, Computer Sc. Dept., U. of Mass, 1989.

[34] A. Podgurski and L. Clarke, "A formal model of program dependencies and its implications for software testing, debugging, and maintenance," *IEEE TSE*, vol. 16, pp. 965–979, 1990.

[35] R. J. Eickemeyer and S. Vassiliadis, "A load instruction unit for pipelined processors," *IBM Journal of Research and Development*, vol. 37(4), pp. 547–564, July 1993.

[36] F. Ricca and P. Tonella, "Construction of the system dependence graph for web application slicing," in *Proceedings. Second IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE, 2002, pp. 123–132.

[37] A. Roth, A. Moshovos, and G. S. Sohi, "Improving virtual function call target prediction via dependence-based pre-computation," *ICS '99*, 1999.

[38] S. Kachigan, "Statistical analysis: An interdisciplinary introduction to univariate and multivariate methods," 1986.

[39] A. Sabelfeld and A. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[40] Y. Sazeides and J. Smith, "Implementations of context-based value predictors," ECE-TR-97-8, University of Wisconsin-Madison, Tech. Rep., 1997.

[41] S. Siegel, *Nonparametric Statistics For The Behavioral Sciences*. NY: McGraw-Hill, 1956.

[42] F. Tip, "A survey of program slicing techniques," 1994.

[43] P. Tonella and F. Ricca, "Web application slicing in presence of dynamic code generation," *Automated Software Engineering*, vol. 12, no. 2, pp. 259–288, 2005.

[44] A. P. W. Masri and D. Leon, "Detecting and debugging insecure information flows," in *ISSRE*, 2004.

[45] W. Masri and A. Podgurski, "Algorithms and tool support for dynamic information flow analysis," *Inf. Softw. Technol.*, vol. 51, no. 2, pp. 385–404, February 2009.

[46] ——, "Measuring the strength of information flows in programs," *TOSEM*, vol. 19, no. 2, October 2009.

[47] M. Weiser, "Program slicing," ser. ICSE '81, 1981, pp. 439–449.

[48] Y. Sazeides and J. E. Smith, "The predictability of data values," *MICRO-30*, 1997.

[49] Y. Sazeides, S. Vassiliadis and J.E. Smith, "The performance potential of data dependence speculation and collapsing," *MICRO*, 1996.

[50] J. Zeng, Y. Fu, K. A. Miller, Z. Lin, X. Zhang, and D. Xu, "Obfuscation resilient binary code reuse through trace-oriented programming," in *CCS*, 2013.

[51] Y. Zhou, L. Wu, Z. Wang, and X. Jiang, "Harvesting developer credentials in android apps," in *ACM conference on security & privacy in wireless and mobile networks*, 2015.

[52] J. Zimmermann, L. Mé, and C. Bidan, "Experimenting with a policy-based HIDS based on an information flow control model," in *ACSAC*, 2003.

[53] J. Zimmermann, L. Mé, and C. Bidan, "An improved reference flow control model for policy-based intrusion detection," in *European Symposium on Research in Computer Security*. Springer, 2003, pp. 291–308.

APPENDIX

A. SPEC CPU 2006 Benchmarks

Table III describes the SPEC CPU integer benchmarks we use.

TABLE III. CINT2006 BENCHMARKS.

Benchmark	Application Area	Description
astar	Path-finding Algorithms	Pathfinding library for 2D maps, including the well known A* algorithm.
bzip2	Compression	Julian Seward's bzip2 version 1.0.3, modified to do most work in memory, rather than doing I/O.
gcc	Compiler	Based on gcc Version 3.2, generates code for Opteron.
gobmk	Artificial Intelligence: Go	Plays the game of Go, a simply described but deeply complex game.
hmmer	Search Gene Sequence	Protein sequence analysis using profile hidden Markov models (profile HMMs).
h264ref	Video Compression	A reference implementation of H.264AVC, encodes a videostream using 2 parameter sets. The H.264AVC standard is expected to replace MPEG2.
libquantum	Physics, Quantum Computing	Simulates a quantum computer, running Shor's polynomial-time factorization algorithm.
mcf	Combinatorial Optimization	Vehicle scheduling. Uses a network simplex algorithm (which is also used in commercial products) to schedule public transport.
omnetpp	Discrete Event Simulation	Uses the OMNet++ discrete event simulator to model a large Ethernet campus network.
perlbench	Programming Language	The workload includes SpamAssassin, MHonArc (an email indexer), and specdiff (SPEC's tool that checks benchmark outputs).
sjeng	Artificial Intelligence: chess	A highly-ranked chess program that also plays several chess variants.
xalancbmk	XML Processing	A modified version of Xalan-C++, which transforms XML documents to other document types.