DITTANY
Dynamic Information and TainTing flow ANalYsis tool.

by

Walid J. Ghandour

User Guide
for DITTANY

AN ABSTRACT OF THE TOOL

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.

Dittany is a tool support for strength-based dynamic dependence analysis on the x86 platform. It involves three 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, 2) an analysis tool that computes the strengths of the identified dependences using information theoretic and statistical metrics applied on their associated values, and 3) a set of perl scripts that parses and analyses the outputs obtained using the profiler and the strength analysis tool.

Dittany is a building block that can be used in different contexts. We have shown 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.

Contents

Chapter 1

DITTANY: Structure, Installation, Build and Usage

In this chapter, we provide information related to the design structure, installation, build and usage of *DITTANY*.

1.1 DITTANY Data Structure

In this section we describe the data structure of *DITTANY*.

1.1.1 ValuePairs

ValuePairs class is used to store pairs of values of source and target instructions. It has two private data members of type *ADDRINT* that are used to hold the values of the source and of the target instructions. It also has the following functions:

- ValuePairs() and ValuePairs(ADDRINT, ADDRINT): Constructors.

- Initialize(ADDRINT, ADDRINT): Used to set the values of the data members.
- sortValuePairs(ValuePairs): Used to sort *ValuePairs*.
- printValuePairs(): Used to print the data members values.

1.1.2 Depend

Depend class stores dependences between instructions. It has the following private data members:

- targ_inst and src_inst: Store the address of the source and target instructions.
- values: A data member of type (ValuePairs, int) map. It is used to store a pair of values taken by the source and target instructions and the number of occurrences of this pair.
- size: The number of occurrences of the dependence between the source and target instructions.

It has the following functions:

- Depend() and Depend(ADDRINT, ADDRINT): Constructors.
- setDepend(ADDRINT, ADDRINT): Set the addresses of the source and target instructions.
- setValues(ADDRINT, ADDRINT): Add a new pair of values of the source and target instructions. It increments the *size* data member. In case this

pair of values was previously taken, we increment its number of occurrences in the *values* map; Otherwise, we insert a new entry in the *values* map.

- printValues(): Print the addresses of the source and target instructions, the value of the size, and the value of each pair and the number of occurrences of this pair.

- sortDepend(Depend): Sort dependences.

1.1.3 InsBufferEntry

InsBufferEntry class holds all information required about an instruction, such as: instruction address, thread ID, routine name, source and destination operands type and values. It has several data members and functions to store and get this information. It also contains a self pointer *InsBufferEntry *next* and a set of type *Depend*, *insrDepSet*, that holds the information about the instructions on which the corresponding instruction depends.

We only describe here the *UpdateDepend* function since it is frequently used in the tool. It takes three arguments: the first one of type *Depend*, the second and the third of type *ADDRINT*. The function first check if the dependence passed using its first argument exists in its dependences set. If found, we only update it with the pair of values passed using the second and third arguments. If it is not found, we insert it and assign to it the pair of values passed.

We track static instructions using an array that holds pointers of type *InsBufferEntry*. We use a function called *findInsBufferEntry* that is used to allocate an *InsBufferEntry* entry for each instruction.

When an instruction is executed, it is instrumented according to its type using the approach described in Chapter **??**, Section **??**.

1.2 DITTANY Flow Structure

In this section we describe the flow structure of *DITTANY*. The tool first reads the command line options, and registers the appropriate functions to be called to instrument instructions. By default all the application instructions are instrumented. The user can specify using the command line options a selected set of instructions to be instrumented, such as starting and ending routine names, a specific path, skipping a specified number of instructions and the upper bound on the number of instrumented instructions.

We allocate to each instruction selected for instrumentation an instant of the *InsBufferEntry* class described in Section 1.1.3. We statically identifies the *ip-dom* of all branch instructions that are going to be instrumented. According to the type of each instruction, we call the appropriate type-aware function that calculates the dependences of the instruction and its target operands, using the dependences of its source operands. For each of the instructions on which the executed instruction depends, we allocate an instant of the *Depend* class described in Section 1.1.2. Pairs of values that represent the outcomes of the source and destination instructions are stored in instants of the *ValuePairs* calss described in Section 1.1.1 using *UpdateDepend* function described in Section 1.1.3.

When The instrumentation terminated, dependences information is logged. The information contains the total number of instrumented instructions, and an entry corresponding to each dependence that contains the IP of the source and target instructions, the total number of its occurrence, the pairs of values and the number of occurrence of each pair.

1.3 Installation and Build Procedures

DITTANY is based on *PIN* version pin-2.10-43611-gcc.3.4.6-ia32_intel64-linux. It was developed and tested on a machine running Fedora 7, x86_64. I was later on installed and built on two machine, one running Linux Mint, and the other Ubuntu 16.04 LTS.

DITTANY can be installed and built like any pintool. First the user needs to untar dittany.tar under *tools* directory in his *Pin* installation. A directory called *DITTANY* is created. The user needs to change directory to the *DITTANY* directory and run make to build the tool.

1.4 User Interface

The tool can be invoked using the following command like any other pintool:

```
pin -t DITTANY.so -KNOB options -- application
```

We provide the following command line options:

- d: Enables data flow analysis.
- c: Enables control flow analysis.
- s: Indicates that the user will specify the routine at which instrumentation starts and the routine at which instrumentation terminates.

- u: Indicates that the user will specify the routine to instrument.
- sRtn: Specifies the start instrumentation routine.
- eRtn: Specifies the end instrumentation routine.
- LoadSrc: Restricts analysis to flows that have a load instruction as the source of dependence.
- LoadTarg: Restricts analysis to flows into load instructions.
- skip: Specifies the number of instructions to skip before starting instrumentation.
- path: Indicates that the user will specify a path of interest.
- sIP: IP of the beginning of the path.
- eIP: IP of the end of the path.
- total: Specifies an upper limit on the number of instructions to instrument.

Tool command line options should be inserted between the tool name and the double dash.

Application stands for the application that the user intends to instrument.

Chapter 2

DITTANY Components

2.1 Introduction

DITTANY is composed of the following components: dynamic information flow analysis, strength computation, scripting tool.

2.2 Dynamic Information Flow Analysis

2.3 Strength computation

2.4 Scripting tool

Bibliography