Dynamic Binary Instrumentation

for Security Analysis of Programs

Neuchâtel, Nov 29, 2024

Daniele Cono D’Elia (@dcdelia)

Meet DBI

Dynamic Binary Instrumentation (DBI) is the process of inserting additional code into a running binary to gather execution information and possibly alter its run-time behavior. DBI does not require modifications to the binary itself.

Why DBI for this seminar?

- very popular in PL, systems, and security research
- can help you gain insights in problems you are studying
- you may use it to implement research solutions

Outline

What we will cover today

1. Introduction to DBI
2. Basic tutorial with Intel Pin
3. Hands-on exercises
4. Usage tips, evasions & wrap-up

What is DBI?

Formal definition

A DBI system is a user-space execution runtime for process-level virtualization.

An unmodified binary executes within the DBI system as it would happen in a native execution, but introspection and execution control capabilities become accessible to DBI users through well-defined APIs.

The system and the program being analyzed share the same address space.

Key features

Some notable features:

- no need for source code
- high degree of compatibility
- can expose accurate real-time insights
- has intuitive primitives with flexible granularity (e.g., blocks, functions)
- may handle obfuscated code

DBI can be an ideal solution for analyzing real-world complex software and environments with unpredictable behavior.

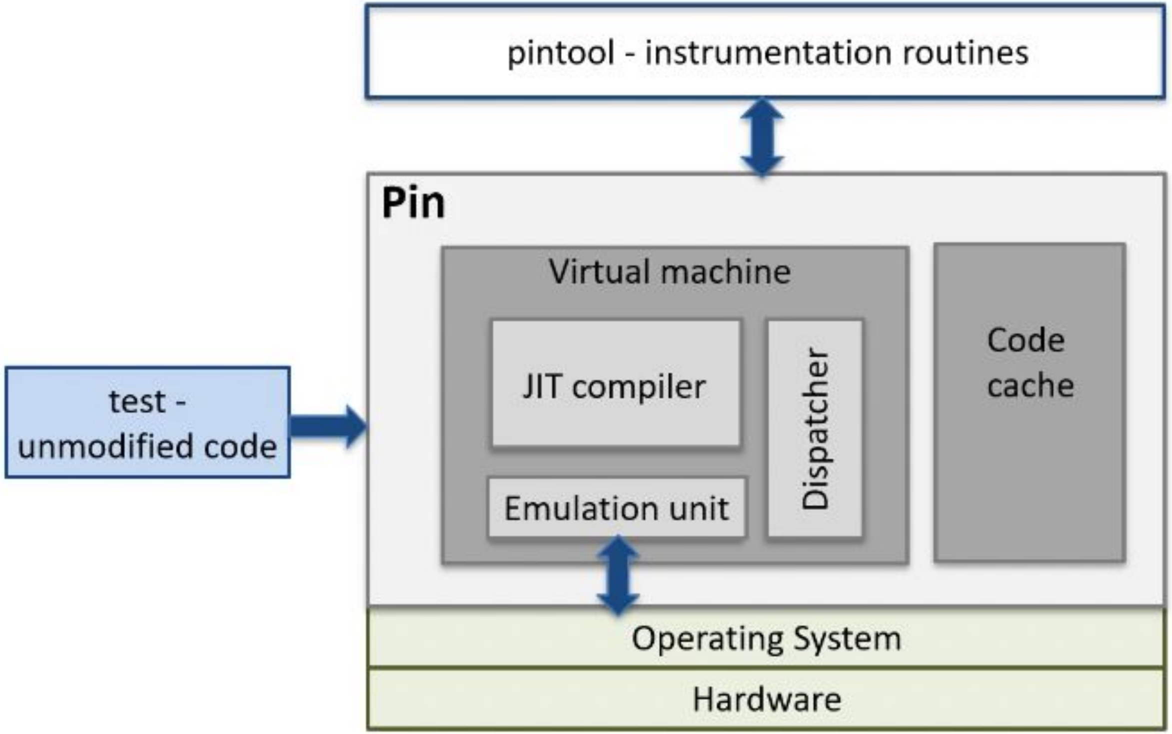
Workflow

High-level functioning of a DBI system:

1. load the target binary
2. pick some instructions for execution (e.g., until next branch)
3. analyze instructions and apply user-specified instrumentation
4. **just-in-time compile** everything into an executable fragment
5. add the fragment to a code cache and **run it**
6. upon fragment end, go back to step 2

Optimizations (e.g., stitch fragments) mitigate intrinsic overheads

Workflow



1. load the binary
2. pick instructions
3. instrument them
4. JIT-compile them
5. cache the JIT'ed fragment and run it
6. go back to step 2

Credits: Alex Balgavy

Differences with static instrumentation

Main alternatives: compiler-based instrumentation, static binary rewriting

Advantages of DBI

- plug-and-play
- support for “dynamic” code
- very flexible instrumentation

Main drawbacks

- performance overhead
- conspicuous traits

Differences with static instrumentation

Main alternatives: compiler-based instrumentation, static binary rewriting

Good use cases for DBI

- profiling behaviors
- debugging & understanding
- hard-to-analyze code

Good use cases for static techniques

- performance testing
- persistent instrumentation

Key DBI techniques

Instrumentation can address multiple aspects of execution. Typically:

- **Instruction** instrumentation (by specific types or addresses)
- **Basic block** instrumentation
- **Function** instrumentation (entry/exit events and arguments)
- **Memory access** monitoring

Additional hooks exist for exception handling, process/thread lifecycle, etc.

Users can analyze the entire userland stack or focus only on program code.

Main DBI systems

Some prominent security uses

SoK work from 2019 reviewed 90+ security papers using DBI for:

- Software protection (30)
- Vulnerability detection (22)
- Malicious software (14)
- Reverse engineering (9)
- Information flow tracking (8)
- Cryptoanalysis (7)

and more. DBI tools can also help with feasibility studies for larger projects.

Writing a DBI tool with Pin

Today we focus on Intel Pin for its broad capabilities and compatibility.

Suggested steps

1. Identify what events you want to monitor/alter
2. Sketch callbacks for analyzing event instances
3. Set up a Pin tool skeleton
4. Register such callbacks with appropriate instrumentation
5. Try them live and continue expanding your tool

Typical instrumentation tasks

Profiling

- Count instructions/blocks/calls/accesses
- Identify targets (e.g., indirect calls)
- Measure execution time
- Log data (e.g., API parameters)

Debugging/understanding

- Validate memory accesses
- Intercept errors
- Track execution context (call stack)

Getting started with Pin

```
#include "pin.H"

int main(int argc, char *argv[]) {
    // Initialize symbol processing
    PIN_InitSymbols();

    // Initialize Pin
    PIN_Init(argc, argv);

    // Add instrumentation function for instructions
    INS_AddInstrumentFunction(InstructionCallback, NULL);

    // Start the program execution
    PIN_StartProgram();

    return 0;
}
```

To simplify the build system, borrow the Makefile rules from the SimpleExamples folder of all Pin releases.

Hands-on

https://github.com/dcdelia/cuso24240006

All you need is a Linux system with g++, build-essentials, and Intel Pin.

Setup instructions are in the GitHub page.

(short link: https://tinyurl.com/cusodbi)

Building our Pin tool #1

We start with a classic task: tracing and counting all executed instructions.

Learning goals

- Instrumentation-time vs analysis-time callbacks
- INS instrumentation: INS_InsertCall with IPOINT_BEFORE (docs)
- Handling execution termination

Building our Pin tool #2

We now move to a classic security scenario: intercepting indirect calls.

Learning goals

- Distinguishing instruction types: INS_IsIndirectControlFlow, INS_IsCall
- Inspecting run-time data: IARG_INST_PTR, IARG_BRANCH_TARGET_ADDR
- IMG instrumentation for whole modules
- RTN abstraction for looking up functions (docs)

Building our Pin tool #3

We will maintain a stack trace and present it to the user at a crash site.

Learning goals

- More instruction types (INS_IsDirectCall, INS_IsRet)
- More run-time data for callbacks
- Intercepting signals: PIN_AddContextChangeFunction and callback
- Validating consistency of retrieved run-time data
- Accessing debug information

Implementation tricks from experience

DBI can be **very expensive**. Design work can mitigate these costs:

- Conducting work at instrumentation (vs. analysis) time is preferable
- Simple analysis callbacks may be inlined
- Try to trade space for speed when possible
- Advanced means (e.g., predicated instrumentation) for intensive tracing

Conspicuous features of DBI

SoK: Using DBI for Security (And How You May Get Caught Red Handed) - ASIACCS’19

What does transparency mean?

For **DBI architects**: support program semantics as in a native execution.

«The further we push transparency, the more difficult it is to implement, while at the same time fewer applications require it.» - DynamoRIO’s creator

For **security researchers**: possibility of <u>adversarial sequences</u>.

Opportunities for attackers

The ASIACCS’19 SoK identifies 7 attack surfaces:

- time overhead
- memory contents and permissions
- leaking a code cache address
- DBI engine internals
- interactions with OS
- exception handling
- translation defects

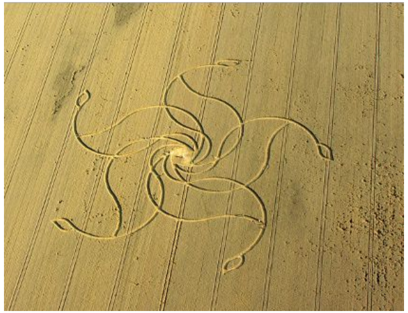
Examples of detections

> Time overhead

- *general slowdown from dynamic analysis*
- *ad-hoc detections*

> Memory contents and permissions

- *extra sections, data patterns*
- *increased memory usage*
- *stress layout for **consistency** (e.g. NX policy, guard pages)*



> Leaking an address from code cache

How to counter DBI evasions

*Theoretically weak, but practically effective: **patch** obvious imperfections.*

Domain experts are aware of the most recurrent possibilities. For example, for malware analysis, we can patch Pin's imperfect handling of `int 2d`.

Some cases are hard: one can then use DBI to conduct a **feasibility** study and then devise later a more complex implementation with other techniques.

What affects a researcher’s choice?

Some relevant factors:

- required instrumentation capabilities
- altering execution can be complex with other technologies
- deployability of the runtime
- tricky behaviors (e.g., sensitivity to time variations)
- characteristics of the adversary

Thanks!