TRANSPARENTLY INTEGRATING DEBUGGING AND DYNAMIC BINARY INSTRUMENTATION

By

Branden Clark

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Approved by the
Examining Committee:

____
Ana Milanova, Thesis Adviser

____
Bülent Yener, Member

____
Vassilis Zikas, Member

Rensselaer Polytechnic Institute
Troy, New York

November 2016
(For Graduation December 2016)

CONTENTS

LIST OF FIGURES . . . iii

ACKNOWLEDGMENT . . . iv

ABSTRACT . . . v

1. INTRODUCTION . . . 1

2. DynamoRIO Overview . . . 3

3. drdbg architecture . . . 4
   - 3.1 Initialization . . . 4
     - 3.1.1 Server interface . . . 4
   - 3.2 API . . . 6
     - 3.2.1 Breakpoints . . . 6
     - 3.2.2 Custom Commands . . . 6

4. drdbg design . . . 9
   - 4.1 Transparency . . . 9
     - 4.1.1 User transparency . . . 9
     - 4.1.2 Client transparency . . . 9
     - 4.1.3 Application transparency . . . 10
   - 4.2 Breakpoints . . . 11

5. Related Work . . . 13

6. Future Work . . . 14

LITERATURE CITED . . . 15

LIST OF FIGURES

3.1 drdbg architecture 4

3.2 drdbg server interface 5

3.3 drdbg command interface 5

3.4 Debug non-zeroing xor instructions API example 7

3.5 Custom command API example 8

4.1 Breakpoint perturbation 10

4.2 Basic block split on breakpoint address 12

ACKNOWLEDGMENT

I would like to thank Derek Bruening and Qin Zhao for working with me and providing their knowledge and expertise of DynamoRIO.

ABSTRACT

With dynamic binary instrumentation becoming increasingly popular we have seen the rise of many applications in analysis, debugging, and control. Dr. Memory[2], for example, aids in the debugging of memory errors by tracking memory and reporting any detected memory errors. Another tool, Godware[12], automates the process of unpacking malware. While tools like these are great, there hasn't been much development into tools that facilitate interactivity and user integration into an environment with familiar debugging primitives. We provide a platform, built on top of the dynamic binary instrumentation system DynamoRIO[1], that allows a user to interactively debug an application, hides DynamoRIO's internals from the user, hides debugging semantics from instrumentation clients, enables integration of debugging primitives with instrumentation clients, and enables improved performance of some debugging primitives.

1. INTRODUCTION

There are several types of instrumentation, source, static, and dynamic. The goal of each is similar, to provide insight and control, and each has their own pros and cons. Source level instrumentation involves modifying the application's source code. The downsides to this are that you need to possess the source code and you have to recompile every time, which can be slow for large projects. Static binary instrumentation involves modifying the application binary to insert your instrumentation. Some benefits to this method over source instrumentation are you no longer require access to the application's source code and you don't have to recompile the application every time. These are great benefits, but since you are modifying the binary on disk, you don't have access to code that may be generated or executed at run time. Dynamic binary instrumentation provides a platform that enables the introspection, insertion, modification, or deletion of code at run time while an application executes.

Tools that are built on these platforms are powerful but they tend not to involve the user. More often than not, the application is analyzed quickly and a report is given to the user upon application termination or upon request. For example, Dr. Memory will print a report at the end of execution or when given a nudge[7]. This allows the user to debug the error after the fact, but it may be much more convenient for the user to debug the error when it occurs. Integrating debugging and dynamic binary instrumentation makes this sort of interactivity possible.

It is easy to think of examples in which a user can benefit from this integration. A user can write a tool that breaks when it sees a non-zeroing xor instruction, which is representative of a common string obfuscation technique. The user can then debug that region of code to extract the plaintext and key, and apply that to decrypt the other obfuscated strings in the application. A user could even write a tool to attach the debugger after jumping into dynamically allocated memory, a typical unpacking technique. The user can then proceed to dump the shellcode or debug it. On the other side, debugging can greatly benefit from dynamic binary instrumentation's fast and powerful features. Certain tasks are very slow in debug-

gers but can be implemented very quickly using dynamic binary instrumentation. An example of this is when a user wants more than four hardware watchpoints. Typical debuggers have to resort to single stepping the application, causing a slow-down orders of magnitude higher than typical execution. Qin Zhao et al.[3] showed how to implement many watchpoints using dynamic binary instrumentation with an average overhead of 2.59x native execution. Speed up of this magnitude will be noticed and greatly appreciated by the user.

In this paper we describe drdbg, our platform for debugging and debugger extensions for DynamoRIO. drdbg provides an interactive debugging interface as well as an API that allows clients to create debugger extensions that can perform the operations we described above. We believe that drdbg will allow clients to better involve the user and create tools that combine the best of interactive debugging and dynamic binary instrumentation.

The main contributions of this paper are:

- An abstract debugger interface for transparent interactive debugging of instrumentation targets. We have provided an implementation of this interface that ties with GDB's remote debugging protocol, but the interface can be implemented with any remote debugging protocol.

- An API that enables clients to extend or implement entirely new debugging primitives, enabling greater semantic and objective insight and control over instrumentation targets.

2. DynamoRIO Overview

To give context to the following sections we will give a brief overview of how DynamoRIO works. A more complete description with details can be found on DynamoRIO's documentation page[6].

DynamoRIO, and other dynamic binary instrumentation systems, can be thought of as application virtual machines. When DynamoRIO detects a new basic block, usually starting with the application's entry point, DynamoRIO copies the basic block into its code cache. A basic block is a sequence of instructions in which each instruction is always followed by the same instruction. Typically, basics blocks will start with a branch target and end with a branch instruction. Before the code is executed it goes through several analysis and instrumentation phases. DynamoRIO performs several other optimizations but the details of those aren't important for this paper.

By itself, nothing that a user finds interesting happens when an application is run under DynamoRIO. Additional analyses and transformations come from clients making use of the DynamoRIO API. DynamoRIO allows clients, software written to make use of DynamoRIO's API, to register callbacks for each of these analysis phases. These callbacks allow the user to analyze, insert, remove, or modify the instructions in the basic block. After the analysis and instrumentation phases are complete DynamoRIO executes the basic block in the code cache. This process is repeated for each new dynamic basic block in the application. The analysis and instrumentation phases are only triggered once for each basic block, when it is first found. Any inserted code will, of course, be executed each time the basic block is executed. Because of this, it is preferable to do as much work as possible during the analysis or instrumentation phases, rather than during every execution of the basic block.

3. drdbg architecture

3.1 Initialization

drdbg can be loaded either at startup by passing the '-appdebug' flag to DynamoRIO, or during execution using DynamoRIO's nudge feature. Either method will load drdbg into the applications address space and initialize it. From here drdbg will create a new thread for communicating with a user interface and insert a breakpoint on the next executed application instruction. Once this breakpoint is triggered drdbg will accept a connection from the user interface and interactive debugging can begin. We have implemented a subset of GDB's[8] remote debugging protocol[9], enabling a user to debug from GDB's familiar environment. This implementation is separate from drdbg's internals, allowing drdbg to be used with any remote debugging protocol provided an interface is implemented.

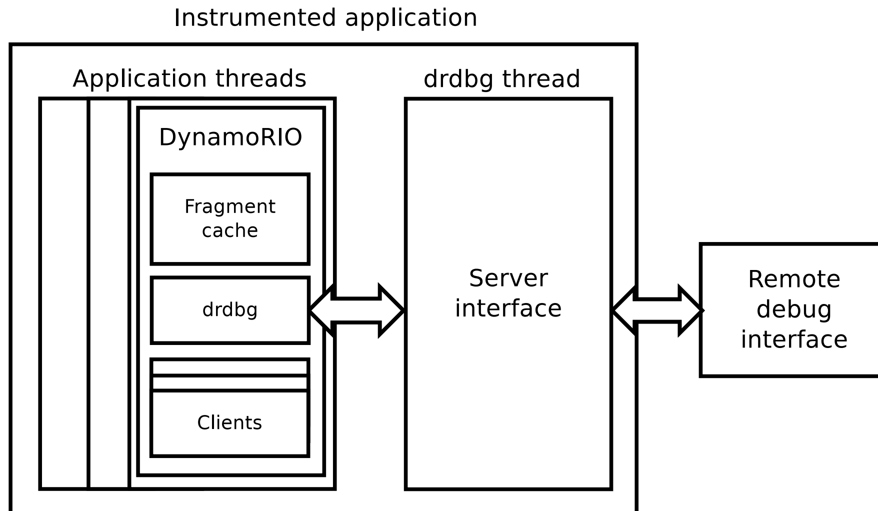

Figure 3.1: drdbg architecture

3.1.1 Server interface

In order to provide the ability to debug using debugger interfaces other than GDB we separated the DynamoRIO logic from the remote debugging protocol parsing. Any remote debugger can be used as long as a parser is written that implements the interface in Figure 3.2. This parser is responsible for communicating with the

remote debugging interface and translating between drdbg's interface and that of the remote debugging protocol.

```
typedef struct _drdbg_srv_int_t {
    drdbg_srv_int_start_t start;   // Start the server
    drdbg_srv_int_accept_t accept; // Accept a connection
    drdbg_srv_int_stop_t stop;     // Stop the server
    drdbg_srv_int_comm_t get_cmd;  // Get command
    drdbg_srv_int_comm_t put_cmd;  // Send command reply
} drdbg_srv_int_t;
```

Figure 3.2: drdbg server interface

drdbg will use these functions to manage the server. The *get_cmd* and *put_cmd* functions are used for pulling and pushing to the server, respectively. Each accepts only one argument, a pointer to a structure, listed in Figure 3.3, which *get_cmd* will write to and *put_cmd* will read from.

```
typedef struct _drdbg_srv_int_cmd_data_t {
    drdbg_srv_int_cmd_t cmd_id;
    void *cmd_data;
    drdbg_status_t status;
} drdbg_srv_int_cmd_data_t;
```

Figure 3.3: drdbg command interface

The *cmd_id* entry specifies which command this structure currently represents, register read, memory write, and single step are a few examples. Each command has its own structure specified for *cmd_data* that must be followed. For example, memory read has a structure that contains the address to read from, the number of bytes to read, and a pointer to a buffer that will contain the memory once it is read. Lastly, *status* is used to indicate success or failure reasons. The server must adhere to the command formats specified in the interface.

3.2 API

Besides the base debug support that drdbg offers we want to make it easily extendable. The API that drdbg exposes enables clients to extend the functionality of drdbg to make the client and debugging process more useful.

3.2.1 Breakpoints

The drdbg API allows a client to interrupt an application at an arbitrary location using *drdbg_api_break*(). This allows a client to be made that performs special analyses or checks on the application and present an opportunity to debug to the user when applicable. Going back to our example from chapter 1, this would allow someone using Dr. Memory to debug a memory error the instant it occurs. Besides being convenient for the user, this is important for situations where the error may not be easy to reproduce. The ability to present a debugging interface to a user at a point deemed special by some arbitrary analysis has a lot of potential. The code in Figure 3.4 is from an example client that wants to break on all non-zeroing xor instructions, in an attempt to identify string obfuscation. This could easily be taken a step further by having the client perform more analysis to determine if this xor really is related to string obfuscation before causing a breakpoint.

3.2.2 Custom Commands

We also provide the ability for clients to register custom commands that can be used through the interactive debugging interface. This enables the user to interact with the clients and any debugging features they might add using the same familiar interface. To do this we make use of GDB's "*monitor*" command, which sends a string to the remote stub.

An example that makes use of this feature is a client that can trace specific instructions or addresses. This client uses drdbg's API to register commands that accept these parameters from the user. If a user wants to see the arguments of the "*cmp*" instruction they found at 0x08048912 they would send something like "*monitor itrace address 0x08048912*". The client will then parse this command and from that moment on will print the instruction at that address along with its

```
static void
callback(app_pc addr)
{
  drdbg_api_break(addr);
}
static dr_emit_flags_t
event_app_instruction(void* drcontext, void *tag, instrlist_t *bb,
                      instr_t *instr, bool for_trace, bool translating,
                      void *user_data)
{
  if (instr_get_opcode(instr) == OP_xor) {
    opnd_t a = instr_get_dst(instr, 0);
    opnd_t b = instr_get_src(instr, 0);
    // Check if non-zeroing
    if (opnd_is_reg(a) && opnd_is_reg(b) &&
      opnd_get_reg(a) != opnd_get_reg(b)) {
      dr_insert_clean_call(drcontext, bb, instr, (void *)callback,
                           false /*no fp save*/, 1,
                           OPND_CREATE_INTPTR(instr_get_app_pc(instr)));
    }
  }
  return DR_EMIT_DEFAULT;
}
```

Figure 3.4: Debug non-zeroing xor instructions API example

arguments. This can be used to defeat an application that is checking a password or license key that that eventually boils down to a “*cmp*” instruction. Once the user identifies this location this client can be used to dump the expected password or license key. Figure 3.5 shows part of a client that implements this example. The client could even implement a “trace by instruction” feature, that allows the user to use a command of the form “*monitor itrace ins cmp*” to trace all cmp instructions in the application. Normally, this would give a lot of output that the user isn’t interested in, but since the user is in an interactive debugging session, the instruction tracing doesn’t have to be enabled until the interesting code is reached. This allows the user to focus on the important cmp instructions, as well as providing a performance boost.

```
drdbg_status_t
cmd_handler(char *buf, ssize_t len, char **outbuf, ssize_t *outlen)
{
    if (!strncmp("itrace", buf, strlen("itrace"))) {
        trace_pc = (app_pc)strtoul(buf+strlen("itrace "), NULL, 16);
        dr_fprintf(STDERR, "trace_pc: "PIFX"\n", trace_pc);
        return DRDBG_SUCCESS;
    }
    return DRDBG_ERROR;
}

DR_EXPORT void
dr_client_main(client_id_t id, int argc, const char *argv[])
{
    [...]
    /* Register command handler with drdbg */
    drdbg_api_register_cmd(cmd_handler);
}
```

Figure 3.5: Custom command API example

This type of user-client interaction comes in handy for clients who perform expensive analysis tasks. The user likely doesn't care about analyzing the application initialization or potentially even large parts of the application. Using drdbg a user can place a breakpoint on a function of interest and wait for it to be triggered. Once the application hits the breakpoint the user can start the client's expensive analysis, and then even disable it once the function ends. Once again allowing the user to quickly hone in on an area of interest and avoid wasted time spent analyzing uninteresting code.

4. drdbg design

4.1 Transparency

4.1.1 User transparency

While one of the primary motivations behind creating drdbg was the current inability to debug an instrumentation target, it is more precise to say that we are currently unable to debug an instrumentation target *effectively*. While it is technically possible to use a debugger to attach to an instrumentation target it won't be very helpful to the user. DynamoRIO drastically changes the application while it executes and this mutated state would be presented to the user instead of what they expect. One major contributor to this issue is DynamoRIO's code cache. DynamoRIO doesn't execute any of the application's code from the application's code section. Instead, when DynamoRIO detects a new dynamic basic block it copies it to a code cache and executes it from there. This is done for performance reasons, and is extremely effective, but if a user were to place a breakpoint on an instruction in the application's code section it would never be triggered because that code is never executed. In addition to this, the PC reported to the user is a code cache address and not a code section address, this can be confusing to a user unfamiliar with DynamoRIO internals. drdbg corrects these inconsistencies and only presents a pure application state to the user, enabling the user to debug the application effectively.

4.1.2 Client transparency

Although we would maintain our benefits over traditional debuggers, we still want to be able to debug more than those applications that are running strictly under DynamoRIO. We want drdbg to be compatible with clients performing their own analyses on the application, and we want to cause as little perturbation as possible to those analyses. For example, debuggers typically implement breakpoints by inserting a trap instruction at the breakpoint target. This changes the code in memory and isn't be representative of the application during a typical run. Figure 4.1 highlights

Pure application code	Perturbed code	Perturbed execution
mov rax, [rbp − 0x8]	**mov** rax, [rbp − 0x8]	**mov** rax, [rbp − 0x8]
mov rdi, rax	**mov** rdi, rax	**mov** rdi, rax
call puts	**int** 0x03	**int** 0x03
	[junk]	[interrupt raised]
		[pc backed up]
		call puts

Figure 4.1: Breakpoint perturbation

the differences in the code of a simple function call and its semantics during run time. Just as we want to prevent a clean view to the user, we also want to present a clean view to clients performing analyses. To resolve this problem for breakpoints we do not use the typical trap breakpoint, but instead use the implementation discussed in section 4.2. While it is true that code is inserted for the clean call, DynamoRIO makes a distinction between application instructions and meta instructions. Because of this, we do not have to worry about clients' analyses confusing our clean call with original application instructions.

4.1.3 Application transparency

Debugging is an inherently invasive operation so we do not make any attempts beyond the ones made by DynamoRIO to conceal ourselves from the application. As part of DynamoRIO's transparency our thread is hidden from the application, and any library dependencies are separated to avoid conflicts as long as DynamoRIO's private loader is used. These things are done primarily to maintain consistency between running an application inside and outside of DynamoRIO, not to act as a sandbox or to be stealthy. A recent proof of concept[10] shows that it is even possible to escape DynamoRIO's control. While it is technically feasible to avoid these problems, DynamoRIO prefers to focus on compatibility and efficiency over anti-anti-analysis.

We are, however, able to avoid several common anti-debugging tricks simply because drdbg is not a typical debugger. On Linux a common anti-debugging trick is to attempt to ptrace[11] yourself with PTRACE_TRACEME, if this fails with EPERM it could be because you are already being traced. Now that you know you are being debugged you can adjust your behavior accordingly. We are not using

ptrace to exercise control over the application so this trick cannot be used to detect drdbg. On Windows, each process has a process environment block (PEB) which holds details about the current process. There are several flags in the PEB that can be checked to confirm whether on not you are being debugged. Two examples are the IsBeingDebugged flag and the HeapFlags entry, both of which are set by Windows during the loading stage. While IsBeingDebugged is fairly self explanatory, the HeapFlags entry is a debugger giveaway because Windows implements a separate debug heap which is used by applications while they are being debugged. There is a flag in the HeapFlags entry that tells whether or not the debug heap is being used. Because we are not using Windows' debug API these flags will not be set as they would be from using a typical debugger, and we can avoid detection.

4.2 Breakpoints

Due to the reasons discussed in 4.1.2 we do not use trap instructions to implement breakpoints, but instead developed a breakpoint model that makes use of DynamoRIO to avoid these issues. Our model uses DynamoRIO's basic block analysis phases to identify the correct region of code and to insert a clean call that allows us to gain control. However, these analysis phases are only run the first time DynamoRIO encounters each basic block and it is possible that an address that a user wants to break on is already in the code cache. If this is the case then we will not detect the basic block in any further analysis phases unless we flush the code cache. Flushing the code cache is usually expensive because any basic blocks encountered from that point onward will have to be re-added to the code cache. To avoid this expense, we only flush the basic blocks and traces that contain the breakpoint address. This way, basic blocks or traces that don't involve the breakpoint address remain in the code cache and will execute as if nothing has changed.

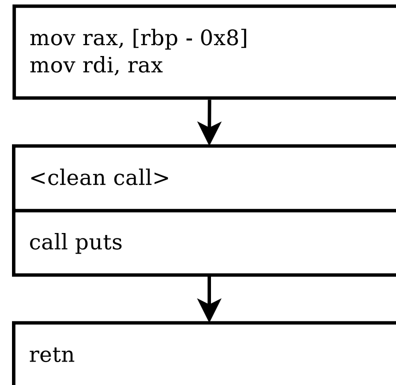

Figure 4.2: Basic block split on breakpoint address

When a basic block that contains a breakpoint address comes back through the analysis phase after being flushed, we split the basic block on the breakpoint address. As shown in Figure 4.2, this gives can give us at most three "basic blocks", the instructions before the breakpoint address, instruction at the breakpoint address, and the instructions after the breakpoint address. If the breakpoint is at the start or end of a basic block then there will only be two "basic blocks", of course. Next, we insert a clean call on the breakpoint address. When execution reaches this point control will transfer to our clean call, allowing us to perform whatever action we wish. We utilize this opportunity to push an event to notify the server thread that a breakpoint has been hit and to suspend the other application threads. Once the server thread picks up the event it will notify the remote debugging interface that a breakpoint has been hit. The user is then able to use the remote debugging interface to debug the application normally.

5. Related Work

Lueck, Patil, and Pereira implemented a similar system for the dynamic Instrumentation platform Pin called PinADX[5]. Their goal was very similar to ours, to enable debugging of instrumentation targets and also to provide a platform for interactive debugging tools. PinADX was implemented as an integral part of Pin, as part of their internal trace fetching mechanism. As a result, PinADX is also closed source, and the design is inherent to Pin's design. Our platform, like DynamoRIO, is open source, but in addition, its design is based on DynamoRIO's API, rather than being integral part of DynamoRIO. This allows our system to be implemented on other dynamic binary instrumentation systems as long as they implement a semantically similar API. We believe that the DynamoRIO APIs we make use of are general enough that this claim is not a reach.

Qin Zhao et al. published EDDI[3], Efficient Debugging Using Dynamic Instrumentation, which uses DynamoRIO to efficiently implement watchpoints. The idea was that hardware has a limited number of watchpoints and when they are all used debugging becomes extremely inefficient. They were able to utilize DynamoRIO to place millions of watchpoints with very little overhead from normal/native execution. This is different from our goal of implementing a general debugging framework for DynamoRIO. EDDI would actually make a great addition to our framework, as we don't provide our own implementation of watchpoints yet.

6. Future Work

We believe that our platform is a great first step for enabling more user-friendly debugging related DynamoRIO tools. While we were successful in allowing a user to debug an instrumentation target using GDB's remote debugging protocol, and in providing an API that allows extensions to be written, we didn't implement as many things as we considered.

Firstly, we support software breakpoints, but not hardware breakpoints or watchpoints. Integrating DR EDDI, or implementing something similar, would provide significant speed increases over standard debuggers when using more than four watchpoints, as well as providing currently missing functionality.

Another important avenue is support for other platforms. While drdbg was designed with cross-platform support in mind, it hasn't been tested. At the very least, the socket code in our implementation of GDB's remote debugging protocol would have to be modified. Since DynamoRIO is cross-platform, there shouldn't be much else in the way of Windows support.

LITERATURE CITED

[1] D. Bruening, “Efficient, Transparent, and Comprehensive Runtime Code Manipulation,” Ph.D. dissertation, Massachusetts Inst. of Technology, Cambridge, 2004.

[2] D. Bruening and Q. Zhao, “Practical Memory Checking with Dr. Memory,” in *Int. Symp. Code Generation and Optimization*, 2011, pp. 213-223.

[3] Q. Zhao et al., “How to Do a Million Watchpoints: Efficient Debugging Using Dynamic Instrumentation,” in *Int. Conf. Compiler Construction*, 2008, pp. 147-162.

[4] C. Luk et al., “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation,” in *Programming language design and implementation* 2005, pp. 190-200.

[5] G. Lueck et al., “PinADX: An Interface for Customizable Debugging with Dynamic Instrumentation,” in *Int. Symp. Code Generation and Optimization*, 2012, pp. 114-123.

[6] Google Inc, Mountain View, CA, USA, “The DynamoRIO API,” 2016. [Online]. Available: http://dynamorio.org/docs/. [Accessed 24-Oct-2016].

[7] Google Inc, Mountain View, CA, USA, “The DynamoRIO API: Linux Deployment,” 2016. [Online]. Available: http://dynamorio.org/docs/page_deploy.html#lin_deploy. [Accessed: 24-Oct-2016].

[8] Free Software Foundation Inc, Boston, MA, USA, “GNU Project Debugger,” 2016. [Online]. Available: https://www.gnu.org/software/gdb/. [Accessed: 13-Oct-2016].

[9] Free Software Foundation Inc, Boston, MA, USA, “GDB Remote Debugging Protocol,” 2016. [Online]. Available: https://sourceware.org/gdb/onlinedocs/gdb/Remote-Protocol.html. [Accessed: 13-Oct-2016].

[10] C. Gorgovan, “Escaping DynamoRIO and Pin,” 2016. [Online]. Available: https://github.com/lgeek/dynamorio_pin_escape. [Accessed: 13-Oct-2016].

[11] M. Kerrisk, “ptrace documentation,” 2016. [Online]. Available: http://man7.org/linux/man-pages/man2/ptrace.2.html. [Accessed: 24-Oct-2016].

[12] J. Bremer, “Malware Unpacking Level: Pintool,” 2012. [Online]. Available: http://jbremer.org/malware-unpacking-level-pintool/. [Accessed: 31-Oct-2016].