A Framework for Robust, Low-Overhead Binary Instrumentation

Amir Majlesi-Kupaei
University of Maryland
Collge Park, USA
email: majlesi@umd.edu

Danny Kim
University of Maryland
Collge Park, USA
email: dannykim32@gmail.com

Kapil Anand
Google Inc.
Mountain View, USA
email: kapilanand2@gmail.com

Aparna Kotha
SecondWrite LLC
Collge Park, USA
email: akotha@secondwrite.com

Khaled Elwazeer
Google Inc.
Mountain View, USA
email: kelwazeer@gmail.com

Rajeev Barua
University of Maryland
Collge Park, USA
email: barua@umd.edu

***Abstract*—We have designed and implemented a binary rewriter called RL-Bin, which can rewrite binaries correctly with low overhead. Existing binary rewriters have their challenges: static rewriters do not reliably work for stripped binaries (*i.e.*, those without relocation information), and dynamic rewriters suffer from high base overhead. Because of this high overhead, existing dynamic rewriters are limited to off-line testing, and cannot be used in deployment. RL-Bin differentiates itself from other binary rewriters by having the capability to robustly rewrite stripped binaries with very low overhead (averaging 1.05x for SPECrate 2017, not including the instrumentation cost, compared to 1.16x overhead for DynamoRIO). The overhead added by RL-Bin itself is negligible and it is proportional to the added instrumentation. Hence, lightweight instrumentation can be added to applications that are deployed in live systems for monitoring and analysis purposes.**

Keywords–Program Analysis; Binary Instrumentation; Static and Dynamic Analysis; Testing and Verification; Program Transformation.

I. INTRODUCTION

There are several reasons why it is desirable to instrument or modify the code that is directly executed in deployment. The applications of instrumentation range from resource monitoring, application performance monitoring, security policy enforcement, vulnerability patching, dynamic information flow tracking, and performance optimization. The code modification can be applied either at the level of source code or binary code.

Binary code is the code that executes directly on the hardware using machine code instructions. Binary code can theoretically be produced from any language, but is typically produced not only from older languages like C, C++, Fortran, and COBOL, but is also often produced from popular modern languages, such as Go, Erlang, Visual Basic, Swift, and Objective C.

Binary code is widespread because it offers significant advantages for two types of code: IP-protected code and high-performance code. First, IP-protected code is code that is sold by companies to outside parties. Second, high-performance programs, such as those in the domains of image processing, financial transactions, machine learning, and scientific codes are often deployed in binary code to ensure the highest execution speed. For the aforementioned application areas, it is often needed to be able to instrument or modify the binary code when the source code is unavailable, such as for third-party binaries. To do so, we need a tool named binary rewriter.

A. Criteria and Trade-Offs in Building a Binary Rewriter

There are *two equally important and necessary criteria that a binary rewriter must have: it must be robust, and it must incur low overhead.* First, a binary rewriter must work for different types of binaries, including those produced by commercial compilers from a wide variety of languages, and possibly modified by obfuscation tools. Second, the binary rewriter must be low overhead. Although the *off-line use of programs*, such as in testing and profiling, can tolerate large overheads, the use of binary rewriters in *deployed programs* must not introduce significant overheads; typically, it should not be more than a few percents [1].

Unfortunately, existing methods cannot modify binary code in a manner that is both robust and low overhead. To understand the reason, let us consider that there are two types of binary rewriters: static vs. dynamic. Static rewriting refers to approaches which take an executable binary program as input, and without running it, produce another (rewritten) binary program as output that has the same functionality as the input program, but is enhanced in some way, for example in improving its run-time, memory use, or security. Dynamic rewriters change the binary code during its execution and modify the binary code in memory, either in-place or in a copy of the code memory.

Static rewriters are not robust. Static rewriters can have very low overhead, but are not robust, meaning that they often do not work for certain types of binaries. As our related work section details, 24% of commercial benign programs had dynamically generated code and 1% had obfuscated code, which are the features that static rewriters cannot handle. Several schemes do not even work for simpler programs with indirect branches whose targets cannot be statically determined.

Dynamic rewriters have high overhead. In contrast, dynamic rewriters are robust but have high overhead, usually ranging from 20% to several hundred percents. Dynamic rewriters are robust because they discover all code at run-time. However, they incur high overhead since most of them maintain a code cache, where rewritten copies of code blocks are stored and executed from. As a result of the above drawbacks, binary rewriters are generally not used in deployment today

 ISBN: 978-1-61208-752-8

on third-party programs, since for those programs, usually no guarantees can be made on how they were compiled.

Dynamic rewriters, such as DynamoRIO [2], copy all the code that executes into another memory region called a *code cache*. The code cache is useful because it ensures robustness; the program still works if a piece of data is mistakenly assumed to be code and rewritten. The reason is that the code cache was changed and the original copy of the code segment is still unchanged.

The overhead of dynamic rewriters is caused by two factors. First, copying the code into the code cache is expensive at run-time. Second, and more seriously, the target addresses of an indirect Control Transfer Instruction (CTI) must be translated at run-time because the locations of code have changed to be in the code cache instead. Such indirect jumps or calls are very common – they mostly arise from return instructions, function pointer calls, and calls to virtual functions in object-oriented languages, such as C++. This translation process is inevitable for DynamoRIO since the original destination address in the program is different from the address of the rewritten code inside the code cache.

B. Robust, Low-Overhead Binary Instrumentation

Consequent to the needs above, we developed RL-Bin. It supports several types of obfuscation, as well as dynamically generated and self-modifying code. As a result, it is robust enough to be used for benign third-party applications. Also, we have designed and implemented several optimizations, so it has very low overhead. We present the following contributions.

- Design and development of the first low overhead dynamic binary rewriter that can handle stripped binaries without relocation or debug information, containing self-modifying or dynamically-generated code or obfuscation.
- An innovative method that tracks the execution of code dynamically by anticipating future control-flow to the new code, and adding instrumentation and breakpoints to process such new code when discovered.
- Using a novel dynamic method to eliminate the overhead of breakpoints, once the new code is discovered.
- Using Just-In-Time (JIT) dynamic analysis of the discovered code and traditional data flow analysis concepts, to find ”Safe” functions and further reduce the overhead by eliminating redundant checks.
- The above design is unlike other dynamic rewriters that translate indirect control transfer addresses to their copies in a code cache.
- The result is the first In-Place dynamic binary rewriter – which does not use a code cache – that combines the robustness and coverage of a dynamic rewriter with the low overhead of a static rewriter.
- Extensive testing and performance comparison with DynamoRIO for SPEC CPU2017 benchmark with over 7 million lines of code in C, C++, and Fortran, compiled with Microsoft Visual Studio, GCC, and ICC compilers.
- Design and implementation of a ”Debugging System in Deployment” as a use case of RL-Bin, which enables the developer to find and patch the errors during execution without sharing debug information with the end-user.

RL-Bin will find use in implementing a variety of applications of binary rewriting. For example, researchers have proposed binary rewriting-based methods for securing untrusted code [3], enforcing control flow integrity [4], implementing software transactional memory [5], self-randomizing instruction addresses [6], profiling tools [7], and taint tracking to prevent sensitive data leaks [8].

The paper is structured as follows: Section II discusses the capabilities and limitations of RL-Bin. Sections III and IV describe the base design of RL-Bin and the optimization methods designed to reduce the overhead. In Section V, we demonstrate the results of our evaluation of RL-Bin and compare them to DynamoRIO. Section VI looks into a debugging and patching system as a use-case of RL-Bin. Section VII describes related work. Finally, Section VIII looks ahead at future work and concludes the paper.

II. RL-Bin Capabilities and Limitations

In this section, first, we list some of the troublesome features that may occur in benign programs. These must be handled correctly since our goal is robust binary rewriting. Additionally, we briefly go over binaries with features for which RL-Bin might fail to instrument properly, most of which are found in malicious applications.

A. Handling Complicating Features

Obfuscation is a technique used to mislead attempts to reverse-engineer the code. Here, we are primarily concerned about control-flow obfuscation, which makes it appear that data is code, or vice-versa. There are publicly available applications and research methods which will control-flow-obfuscate a binary application to protect the binary from reverse-engineering, such as the Binary obfuscation project tool [9], and the work by Popov et. al [10].

RL-Bin supports two types of obfuscation techniques that are problematic for binary rewriters: (i) unconditional to conditional branch flow obfuscation, (ii) exception-based obfuscation. In the unconditional to conditional branch obfuscation, an unconditional branch is replaced by a conditional branch, one of its targets is never taken. Instead, the never-taken path contains data, rewriting which will break the program. Another technique is exception-based obfuscation. In this method, a change of control flow is achieved without a CTI, using exceptions instead. For example, the program can deliberately trigger a divide-by-zero exception by using a zero value in the denominator. The program may also register a custom exception handler, which can redirect to any arbitrary location in the program.

Dynamically-Generated Code is common in binary applications. It is mostly used when executing user scripts or any script coming from external sources. Another instance of dynamically-generated code is packed code. Unlike dynamic rewriters, static rewriters cannot disassemble such dynamically generated code.

Self-Modifying Code is similar to dynamically generated code with an important difference: the addresses into which dynamically generated code are stored may already contain instructions that have been executed during the program. This modifies the program's code at run-time.

B. Limitations of RL-Bin

RL-Bin is capable of analyzing and instrumenting most of the common commercial binary files which do not have

relocation information, and may have obfuscated, dynamically-generated or self-modifying code. However, RL-Bin is not designed to support adversarial binaries, which can deliberately use methods to prevent their examination by a binary rewriter or a debugger. We will go over certain types of behavior in adversarial binaries that can cause problems for the binary rewriter.

(i) Verifying the memory image by using a checksum. Some adversarial binaries compare the checksum on their memory image against a previously calculated checksum to make sure that the program is not altered by debuggers. The goal is not ensuring integrity, but defeating debuggers. In most commercial binaries, developers know that many users may use debuggers on the software which will not work with such binaries. *(ii) Disabling the debugger.* Binaries can check the presence of a debugger and can try to disable it. As mentioned before, commercial binary applications are intended to support debuggers and binary rewriter can handle them properly. *(iii) Modifying breakpoints inserted by the debugger.* Adversarial binaries attempt to remove breakpoints inserted by debuggers, which can interfere with the operation of rewriters and debuggers. This behavior is limited to only adversarial binaries.

III. Base Design

In this section, we describe the base unoptimized algorithm that is used by RL-Bin. This algorithm has very high overhead (approximately 5x to 10x the run-time of the un-instrumented program for SPEC CPU2017 benchmarks) but demonstrates the correctness of the method.

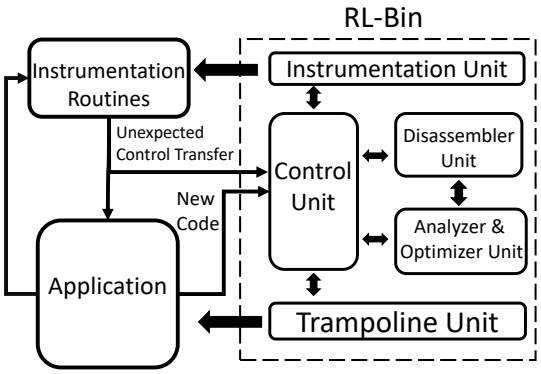

Figure 1. RL-Bin System Overview

The components of RL-Bin are shown in Figure 1. The Control Unit keeps the state of the application and manages other units. The Instrumentation Unit creates and manages instrumentation routines. The Trampoline Unit is responsible for efficiently placing trampolines in the original code to redirect execution to the instrumentation routines.

A. RL-Bin Baseline Algorithm

The main intuition behind RL-Bin is to add instrumentation at run-time that monitors the discovery of the new code. To discover code, our method assumes that a block of memory is code only if we discover an actual control transfer to it during run-time. Our **purely dynamic** disassembly method will begin at the start of a memory block (whose address we call START) once it is proven to be code and follows non-control-transfer instructions one after another, which are all discovered to be code until it reaches a control transfer instruction. Whenever the method reaches a CTI, if that CTI can have more than one possible target, the method ensures that some instrumentation is triggered when the actual target becomes known later during the same run.

Some terminology: All instructions that change the control-flow behavior of a program, such as branches, jumps, and calls, are called *Control-Transfer Instructions (CTIs)*. A *direct CTI* is a CTI whose target is specified by an immediate constant in the instruction. Direct CTIs can be unconditional or conditional. An *indirect CTI* is a CTI whose target is specified in a register or memory location and hence is usually unknown statically.

Here are the steps in RL-Bin's **Disassembly Routine:**

1) Add entry point to the list of instructions to be discovered, let us name it D.
2) Pick an instruction I from list D.
3) Mark the address of instruction I as discovered in the disassembly table.
4) If instruction I is a non-control-transfer instruction,
   4.1. The next instruction must be code as well, so we add it to list D if has not been disassembled before.
5) If instruction I is an unconditional direct CTI,
   5.1. It has only one possible constant target (*i.e.*, it is a direct jump), so we can infer that the target is code as well, so we add the target to list D and disassembly continues from there.
6) If instruction I is a conditional direct branch, (see Figure 2 as an example)
   6.1. We cannot assume that its target (T) and fall-through (F) addresses are both code. As discussed before in Section II-A, because of conditional branch obfuscation, only one of the target or the fall through might be code, but not necessarily both. Hence we insert hardware breakpoints at both the target and fall-through addresses (T and F).
   6.2. Register a custom exception handler for handling these hardware breakpoints. Particularly, when either one of them is executed (say T),
      i) It will register that memory location as code in the disassembly table.
      ii) Then it removes hardware breakpoints at both T and F. (The reason that hardware breakpoints are removed from a block after it is executed is that in most ISAs, only a small number of hardware breakpoints are allowed at a time. In the case of x86, there is a limit of four hardware breakpoints that can be set at a time.)
      iii) Adds trampoline at START (see trampoline (1) in Figure 2), which will transfer to instrumentation routine that adds back the hardware breakpoint at the non-executed address among T and F (say F) (If the code is executed from START again, we do not need to disassemble the code from START again, but just insert the hardware breakpoint at F when at START.)
      Note: In the case of x86, if there are more than four non-executed addresses in the function, extra trampoline(s) will be placed in the middle of the function to remove hardware breakpoints from previous addresses and insert them on the following addresses.
   6.3. Later, as an optimization, if the handler at F also executes, remove the hardware breakpoint, as well as the instrumentation at START. This leads to zero overhead in

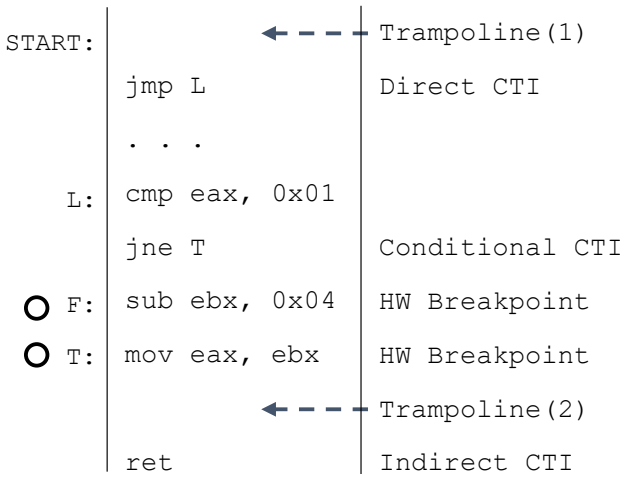

Figure 2. Disassembling a Memory Block

the steady-state after T and F are both proven to be code.

7) If instruction I is an indirect CTI,
   7.1. Insert trampolines to an instrumentation routine (see trampoline (2) in Figure 2), just before the indirect CTI to the instrumentation routine which,
      i) Computes the target upon reaching that point.
      ii) Add it to the list D, if it is not disassembled before.
   (The target of indirect CTIs needs to be checked every time because it can change every time the instruction is executed; hence, our trampoline and instrumentation will remain in place to check the target of indirect CTIs to discover new code and handle unexpected control flows.)
8) If D is empty, then exit, otherwise go to Step 2.

The above method works for dynamically-generated code without a special case, since it tracks the CTI into the dynamically-generated code just like any other CTI. It also handles unconditional to conditional branch obfuscation as described above. However, the method needs additional components to handle self-modifying code and exception-based obfuscation. These will be described in the subsections below.

B. Handling Self-Modifying Code

Self-modifying code is handled as follows.

1) To check whether the code has modified itself, write-protect the pages that contain code, so any write to these pages will cause an exception.
2) Register the exception handler to:
   2.1. Check the addresses which are being written.
   2.2. If they have previously been discovered as code, remove those entries from the disassembly table. (As a result, the newly written code will be treated the same as the code which has never been seen before.)

The above method is very high overhead and it needs to be optimized. The main overhead comes from the fact that every write to the code segment will cause an exception. Such writes will happen if data is stored in the code segment and is written to by the program. To reduce the overhead, we use the following scheme. We add instrumentation code around memory store instructions that trigger the exception for the first time. The instrumentation will turn off write protection, check the addresses being written to, and turn back on write protection after the memory store. In this way, stores to data locations in the code segment will never trigger an exception more than once. As a result, only a small portion of memory store instructions (those that write to the code segment) will be surrounded by our added instrumentation.

C. Handling Exception-Based Obfuscation

This obfuscation happens when an instruction that is not a CTI is used to transfer control of the program. As an example, a divide instruction that deliberately triggers an exception can be used as a CTI. As a result, the memory location following the divide instruction may never be executed. It may contain data and not code. To handle exception-based obfuscation, we follow the following method.

1) Create a stub for every exception handler that is registered. When an instruction triggers the exception it will execute our instrumentation before the actual exception handler.
2) Disassembly routine must stop disassembly at every instruction that can cause an exception that has been registered so far. (In the common case no such exceptions will be registered, thus the overhead will be minimal.)
   2.1. If such an exception causing instruction is found (in Step 4 of the baseline algorithm), put a hardware breakpoint on the instruction that immediately follows it.
   2.2. After hitting the breakpoint, remove it and start discovering code from that location. (This method ensures that no data is mistakenly assumed to be code.)

Using the algorithm in this section, more and more code is discovered during run-time. This method will ensure that not a single instruction can be executed without first being observed by our binary rewriter, even if the instruction has been generated dynamically or through self-modification. Also, in case there is obfuscation, we would never instrument data inside the code segment since we instrument only the locations that contain code that has been executed during run-time.

D. Handling Multi-Threaded Applications

By the advent of multi-core processors, multi-threaded applications have become very common. As a result, every binary rewriter must handle such applications. The main issue in multi-threading is to make sure that the data structures that are shared between threads are being used correctly. Specifically, they should not be used by a thread while simultaneously being updated by another thread. To avoid the problems regarding concurrent access to RL-Bin data structures, each thread must acquire the lock before being able to modify RL-Bin internal data structures. During this modification, no other threads are allowed to access the same data structure.

IV. Optimizations

This section presents the optimization techniques used in RL-Bin to reduce the overhead. The effectiveness of each optimization will be discussed in Subsection V-C.

OP1. Conditional Branches

As was described in Step 6.3 of the baseline algorithm, if at any point both outcomes of a conditional branch are registered as code, then the instrumentation and hardware breakpoints at that branch can both be removed. In the steady-state, the checks before most direct conditional branches are removed.

 ISBN: 978-1-61208-752-8

OP2. Predicting the Target of Indirect CTIs

The baseline algorithm in Step 7, instruments every indirect CTI to compute its target at run-time and register it as code. This overhead can be reduced by optimization with the following intuition: indirect CTIs usually transfer the control to one of a few constant targets. We will replace this check with a check which takes less time.

As an example, let us assume that function *foo()* is being called from three different call sites. So, the return instruction of the function will return to the instruction after one of these call sites. First, the target will be checked against the most frequent call site. If it matched, the indirect CTI can safely transfer the control flow back to the call site. The same idea would be done for second and third call sites. In the end, if none of the previous checks were true, we would refer to the disassembly table to check whether the target of indirect CTI has been discovered as code before.

There is a trade-off between overhead and the number of frequent call sites that need to be checked before referring to the disassembly table. We use heuristics on the frequency of each target to determine the optimal number of checks.

OP3. Function Cloning

It is often the case in programs that a small function is being called frequently from a call site. The intuition is to remove the check needed before the return instruction (indirect CTI) to the call site. During Step 7 of the baseline algorithm, we selectively clone functions to reduce the overhead and remove the checks needed before their return instructions.

In this method, the function is cloned so that no check is needed if called from that specific call site. First, the function is copied to a new location. The call instruction is modified to a direct jump to the new location. As a result, no return address will be pushed on the stack. Also, the return instructions in the function are replaced by direct jumps to the instruction after the call site.

Again, we face a trade-off here. If we clone every function, it would lead to excessive code bloat. Thus, we must clone functions selectively only for the call sites when doing so will reduce the overhead significantly.

```
Set I = Instructions in the function
Set C = Set of Safety Conditions(Called Functions)
 1 bool Is_Safe(Address Entry_Point)
 2   Set W={Entry_Point} //Insts waiting to be checked
 3   While(W ≠ ∅)
 4     pick inst from W
 5     if(inst ∈ P) return false
 6     if(inst ∈ Call_Instructions)
 7       add Dests(inst) to C
 8       add Next(inst) to W if(Next(inst) ∉ I)
 9     else add Dests(inst) to W if(Dests(inst) ∉ I)
10     if(stack_height ≠ value assigned before)
11       return false
12     else
13       assign stack_height of Dests(inst)
14     if (inst is an indirect write)
15       if(Write_Address(inst) = Return_Address)
16         return false
17     remove inst from W and add it to I
18   Let c ∈ C , if(Is_Safe(c) = false) return false
19   return true
```

P1 = Set of indirect branch instructions

```
jmp dword ptr [eax*4 + 0x0c]
```

P2 = Set of instructions that modify the stack pointer to a value that is statically unknown.

```
add esp, eax
mov esp, dword ptr [ecx]
```

Not including

```
add esp, 0x4
```

(Added value is constant)

P3 = Set of instructions that write to an indirect address which may or may not be the return address of the function

```
i.e. mov dword ptr [eax], 0x3c
mov dword ptr [esp + ebp*4], eax
```

Not including

```
mov dword ptr [esp + 0x4], eax
```

(Check whether esp+0x4 points to return address)

$$P = \bigcup_{i=1}^{3} Pi$$

Figure 3. The Algorithm to Determine Safety of a Given Function. (None of the instructions in the set P that is defined on the right side, are allowed in a "Safe" function. Dests(inst) returns the targets of CTIs and for non-CTIs, returns the next instruction.)

 ISBN: 978-1-61208-752-8

OP4. Optimizing White-Listed Modules

It often happens that applications load dynamically shared libraries during their execution and then execute functions from them. In most cases, these DLLs are part of the kernel or they are part of the standard library provided by the programming language. It is possible to optimize away the checks needed for some of these DLLs.

The interaction between the main module of the program and the shared libraries happens by calling a function exported by the library. The control will be sent back to the main module after the execution of the function. The only exception is when the library performs a callback and calls a function from the main module. DLLs are analyzed and their callback functions are discovered. If the behavior of the functions and the callback values can be determined before execution, then the analyzed DLL will be white-listed and checks in that module will be optimized away.

OP5. Detecting "Safe" Functions

The most common indirect CTIs are return instructions. The overhead of the checks before return instructions, checks added during Step 7 of the baseline algorithm, can be further eliminated when the function has certain properties. A "Safe" function, can be proven that it cannot modify its return address, hence the return instruction always returns to the instruction after the call site.

We outline in Figure 3 our Just-In-Time (JIT) analysis algorithms, by which the safety of many functions can be established before their execution. For such safe functions, the instrumentation before the return instruction can be removed. The intuition behind the algorithm is to determine the exact addresses of the memory locations on the stack that will be modified by the instructions within the function. If the return address is not modified, then the function will return to the original call site.

"Stack Height" for every instruction, is defined to be the difference between the value of the stack pointer at the entry point of the function and the value of stack pointer at that instruction. For example, a push instruction will reduce the "Stack Height" by four. If the function does not contain any of the instructions defined in Figure 3 as set P, the "Stack Height" of all instructions can be determined before the execution of the function. If there are more than one control flow paths from the entry point to a given address, and "Stack Height" is not the same between different paths, we declare that function as not "Safe" and do not optimize it. This rarely happens in benign code.

The algorithm will determine the "Stack Height" of each instruction and based on the "Stack Height", will determine whether an indirect write rewrites the return address of the function. We also create a list of functions that are called from this function and put them in set C. Later on, after disassembling all instructions in the function, we check the safety of all the functions in set C. If any of the called functions are not safe, the current function will be declared not "Safe". If all the aforementioned checks showed that the return address cannot be modified, the function will be declared "Safe". Note that the algorithm above will be executed only once for each discovered function, thus there will be no overhead in the steady-state.

OP6. Using Data-Flow Analysis to Find "Safe" Functions

OP5 algorithm does not cover some functions, because writing to global or static data, which is not stored on the stack, is frequently done through indirect addressing.

If there is a write to an indirect address, we need to make sure it does not overwrite the return address of the function. Most of the indirect write instructions that write to the stack use stack-derived registers as the base register (in x86, these are esp and ebp registers). So, if the base register is not stack-derived or it is not a copy of these registers, then it cannot modify any value which is previously stored on the stack. As a result, we must ensure the base register is not derived from the stack pointer.

We define the term PNSD, which is short for "Provably Not Stack Derived". If a register value is PNSD, it means that it can be proved during run-time analysis that the current value in the register is not derived from the stack pointer. An indirect write instruction which uses a PNSD register can never write to the stack. We use traditional data-flow analysis to identify all the different definitions that can reach the base register in the write instruction. If all of the definitions of the base register are PNSD, then the base register is also PNSD.

As it is demonstrated in Figure 4, we modify the algorithm in the previous section to check for PNSD variables when there is an instruction, which stores the value to an indirect address. Again, note that the analysis above will be done only once for each discovered function, thus there will be no overhead in the steady-state.

V. Evaluation and Results

We have completed and tested a fully optimized prototype of the above method. Most of the code is written in C++, while there are some functions which are written in x86 assembly, for the sake of optimization. Our experiments are done on a

```
5   if(inst ∈ P) return false
5′  if(inst ∈ P3)
5″  |  if(!Is_PNSD(base register))
5‴  |  |return false
```

P3 = Set of instructions that write to an indirect address which may or may not be the return address of the function.

$$P = \cup_{i=1}^{2} Pi$$

```
i.e. mov dword ptr [eax + 0x38], 0x3c
```

Is_PNSD(eax) returns true if register eax is PNSD

Figure 4. Algorithm Modification to Cover Indirect Write Instructions with PNSD Base Register.

system with Intel Core i7, 3.33GHz CPU with 12 Mb cache and 24.0 GB DDR3 memory on 64-bit Windows 10 OS. We chose the Windows Operating System since most commercial binaries are developed for Windows.

In our experimental setup, we used the SPECrate 2017 Integer and Floating-Point with their reference data sets. SPECrate Integer has 10 benchmarks and all of them are included in our testing. However, we could evaluate 10 out of 13 benchmarks in SPECrate Floating-Point. The other three benchmarks could not be compiled for 32-bit x86 Windows machines, thus *fotonik3d_r*, *cactuBSSN_r*, and *cam4_r* were excluded from the set. This is because our current implementation is for 32-bit Windows binaries; 64-bit binaries can be supported in our theory but are not implemented yet.

Also, we compiled the binaries with three different compilers; Microsoft Visual Studio, GCC, and ICC. The overhead reported for each benchmark is the average of the overhead for binaries compiled with these compilers. In the case that a benchmark could not be compiled with a particular compiler, that compiler is not included in the average.

Comparison with DynamoRIO: Among different dynamic rewriters available, we compared RL-Bin to DynamoRIO. The reason is that DynamoRIO is designed for efficient binary instrumentation. Based on the previous studies [11] [12], Pin and Dyninst have higher overhead in comparison to DynamoRIO.

A. Performance Without Instrumentation

The goal of RL-Bin is to perform only light instrumentation efficiently. Although it can be used to perform heavy instrumentation, such as basic block counting, no binary rewriter can deliver low overhead for such instrumentation, because the added instrumentation itself is heavyweight. Hence such instrumentations are not good use-cases for RL-Bin, whose main motivation is low run-time overhead in deployed code. As a result, we measured the performance overhead of applications running under binary rewriter without added instrumentation. This overhead should be low for use-cases of RL-Bin.

As it is illustrated in Figure 5, RL-Bin outperforms DynamoRIO by a huge margin. In this Figure, A run-time of 100 is the run-time of the original unmodified program without rewriting. (The overhead shown as 107, means the overhead added by the rewriter is 7% without any instrumentation.) In fact, the overhead of DynamoRIO is 1.06x and 1.26x for SPECrate 2017 Floating-Point (Figure 5 (a)) and Integer (Figure 5 (b)) benchmarks respectively (1.16x or 16% on average), whereas the overhead of RL-Bin is 1.015x and 1.09x for the same benchmarks (1.05x or 5% on average). The reason for higher overhead in Integer benchmarks is the higher number of indirect CTIs compared to Floating-Point benchmarks.

B. Performance with Instrumentation

The next experiment measures the overhead added by the rewriters when instrumenting the application to count the number of external calls from the application module to other DLLs. This particular instrumentation is used because the number of locations that need to be instrumented is relatively low. Hence, it is a good use-case of RL-Bin to perform light instrumentation with very low overhead. Figure 6 shows the overhead of RL-Bin ranges from 5% to 130%, with an average of 25% compared to DynamoRIO which

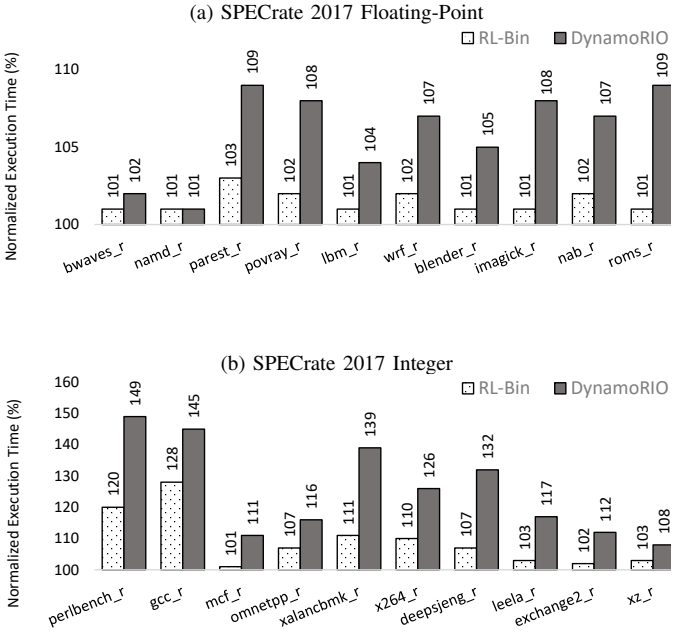

Figure 5. Normalized Run-Time of Rewriters Without Added Instrumentation for SPECrate 2017.

has 56% average overhead (ranging from 4% to 187%). Our experiment demonstrates that RL-Bin can be successfully used to add instrumentation with fairly low overhead compared to DynamoRIO.

C. Optimization Effectiveness

To show the contribution of each optimization method proposed in Section IV, we measured the overhead of SPECrate 2017 Integer with different optimization levels. Figure 7 shows the overhead with six different optimization levels. The overhead is expectedly large (10.25x for *perlbench_r*) without any optimization. Optimizing conditional branches (OP1) will bring the average overhead from 7.24x to 2.93x. Adding target prediction for indirect CTIs will reduce the overhead of remaining checks, thus the average overhead will be 2.02x with OP1+OP2. White-listing modules and cloning functions (OP4 and OP3) will remove lots of the added overhead for checking the target of indirect CTIs and will bring down the average overhead to 1.22x. The last set of optimizations (OP5 and OP6) detect safe functions and remove the check before the return instruction in such functions. Thus, boosting the overhead to just 1.09x on average for SPECrate 2017 Integer benchmarks.

D. Instrumentation Robustness in Commercial Applications

Our last experiment is designed to demonstrate that RL-Bin is robust enough to handle commercial multi-threaded applications that contain dynamically generated and self-modifying code, as well as obfuscation. We aimed to show that RL-Bin fully instruments the binary and it achieves full code coverage, meaning that no instruction is executed without being monitored by RL-Bin. The number of dynamically executed instructions was measured by instrumenting every basic block of the application to add the size of the basic block to the total count.

We tested three popular Microsoft Office tools; Word, PowerPoint, and Excel as well as Adobe Reader, and Apache

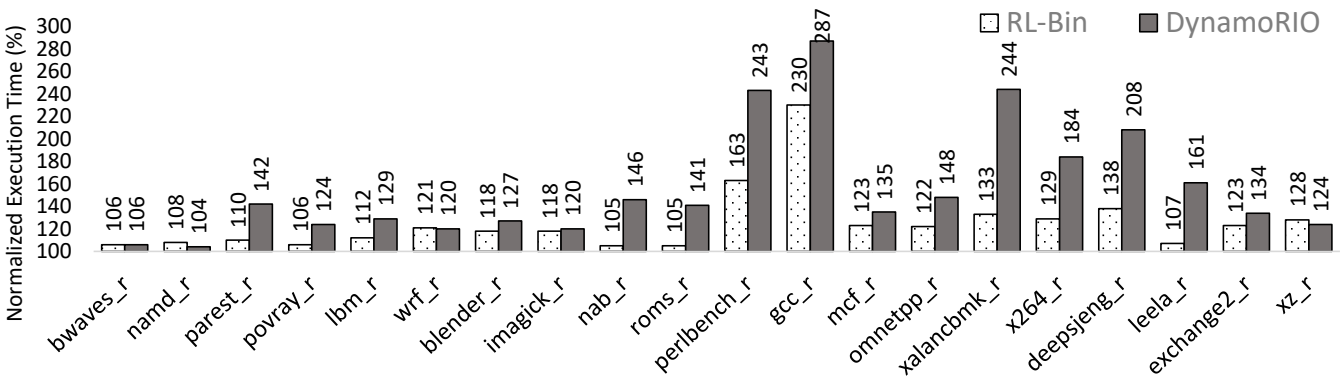

Figure 6. Normalized Run-Time Overhead of Rewriters with Added Instrumentations to Count External Calls for SPECrate 2017

Web Server. In our experiments, in order to have dynamically generated and self-modifying code, we opened documents that contained VBA code in Microsoft Office and JavaScript in Adobe Reader. Apache Web Server heavily uses multi-threading, so this application would appropriately stress test the multi-threading capabilities of RL-Bin.

For commercial programs, we did not measure the overhead, since interaction with users and other uncertain factors, make them unacceptable as benchmarks for measuring the overhead, introduced by RL-Bin. Instead, SPEC CPU 2017 was used for measuring overhead, since they are standardized benchmarks without user interaction, making them suitable for run-time measurement.

The measurements on the number of dynamic instructions were done with both RL-Bin and DynamoRIO. The results showed that the numbers are the same for every application in the set, thus proving that every single instruction is counted by RL-Bin and full code coverage is achieved. As a result, proposed optimization techniques do not result in any loss of coverage, verifying that RL-Bin instrumentation is robust and accurate.

VI. A Use Case: Debugging in Deployment

Making sure that the application is running flawlessly is one of the most arduous tasks in the software development process. In practice, it is often the case that programs face run-time errors, or show unexpected behavior. The main reason is insufficient data sets to test different scenarios. End-user systems will have different resources, and configuration. An error may arise only in certain execution platforms, and never come up in development tests. As a result, debugging is needed even after the development process.

Now, consider the following scenario. The developer has released the software to the end-user, but there is a bug in the software which only happens in the end-user system. The developer cannot reproduce the error in the development environment. There are two existing methods to solve this problem. First, the program may be executed with the presence of a debugger to find where the issue happens. However, almost all commercial binaries are stripped of their debug information to protect their code from being reverse-engineered. As a result, this solution is impractical and the developer will not share debugging information with the user. Another solution is to generate an error log whenever the application crashes and send it to the developer. The log file may contain the current stack and the value of certain attributes of the program. This may be useful to learn more about the issue, however, it is too general, the developer will need extra information. In addition, neither of the methods above would patch the code and solve the issue. Even if the error is found, the user needs to wait for the next release of the application which may take a long time. If the bug is a security concern, it is crucial to patch the program as soon as possible.

Our solution takes as input debugging information of the program and any arbitrary instrumentation that the developer wants to put in the program. We recompile RL-Bin to use the information of the debug file and generate instrumentation that will be inserted in the target application. Based on the debug file, RL-Bin would know where to instrument. The modified

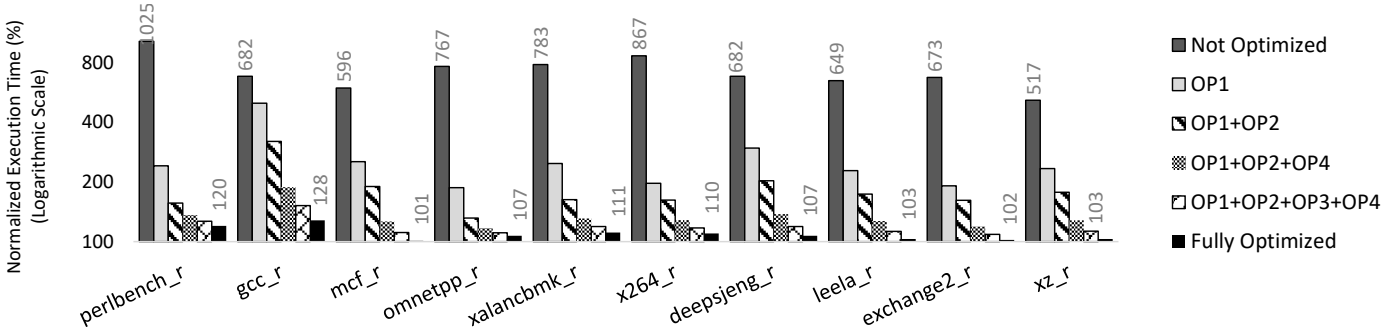

Figure 7. The Contribution of Optimization Methods in Reducing Overhead of RL-Bin for SPECrate 2017 Integer Without Instrumentation

version of RL-Bin, the dynamic debugger, will be sent to the end-user. Added instrumentation will monitor execution and send requested information to the developer. Thus, enabling the developer to pinpoint the problem and fix the issue. This dynamic debugger does not reveal debugging information to the end-user. Only recompiled RL-Bin is sent to the end-user system and the debug information file never gets exposed. Another advantage is that the code can be patched dynamically when the binary is being executed. This is crucial for certain service applications which need to be responsive all the time.

As proof of concept, we developed a simple version of our dynamic debugger. This prototype is capable of parsing PDB file format which stores debugging information of the programs compiled with Microsoft Visual Studio. Our debugger will instrument the program to monitor it during its execution. As an example, instrumentation was added to report the maximum value of the first argument passed to ten random functions. (The purpose of our test was to measure the overhead. The monitored functions and monitoring method depend on the developer and may vary case by case.) Our test results showed that the average overhead was just 6.1% for SPECrate benchmarks, which means that added instrumentation added little extra overhead in comparison to 5% overhead for binaries without instrumentation according to Subsection V-A. This low overhead makes RL-Bin practical for use in deployment.

VII. Related Works

Binary rewriting is a well-researched field of study and during the past thirty years, there have been several major rewriters developed to address the specific needs of the community. [13] thoroughly covers existing works in full depth. In this section, we briefly review existing static and dynamic binary rewriters and compare them against RL-Bin.

RL-Bin [14] represents a very early snapshot of this project published in a non-archival workshop. The current paper is extensively different from [14] by providing a more in-depth analysis, formal definitions of algorithms, more thorough evaluations and experiments that were not part of the earlier paper. As a result of these changes, 72% (about seven out of ten pages) of the material in the current paper is new and never published before.

In particular, [14] presented the initial version of RL-Bin which had high overhead, around 2.5 times the overhead of the current version. The new version has introduced new optimization techniques such as indirect branch target prediction, white-listing external modules, and extending the coverage of safe functions by defining PNSD variables, all of which have helped to achieve overhead of less than 5% on average, indicating that current version can practically be used in deployment. In addition, the earlier paper did not have methods for handling exception-based obfuscation, multi-threaded applications, and self-modifying code. Hence, previously it was only tested for single-threaded applications not containing self-modifying code or exception-based obfuscation. The current version has reached a level of maturity and robustness that can be used for all benign stripped commercial binaries.

A. Static Binary Rewriters

Currently, lots of static rewriting solutions are available including [15]–[19]. SecondWrite project [15] aims to recover compilable source code from binaries, initially output as LLVM IR, which could then further be compiled into rewritten executable code. ATOM [16] provides a flexible interface for code instrumentation which helps in the development of program analysis tools. Diablo [18] aims to provide a framework for link-time program transformation with whole program optimization and instrumentation. Dyninst's version 2007 [12] is an in-place static binary rewriter aiming to provide low-overhead instrumentation capability. Pebil [19] is another static binary rewriter focused on achieving efficient binary instrumentation by using function-level code relocation for inserting control structures.

Static rewriters, including all of the above, face significant limitations due to the lack of run-time information when trying to disassemble and instrument the binary. The first limitation is that they cannot disassemble dynamically generated or self-modifying code. The reason is that these codes are not available before the execution of the program. This will lead to incomplete code coverage.

Dynamically generated code is quite common in benign applications. In a recent study [20], it was observed that 29 out of 120 benign applications contain dynamically generated code, which is used for supporting the execution of user scripts. This means that implementations of security policies that use static binary rewriters would fail for 24% of applications.

The second limitation of static binary rewriting arises from the fact that some benign programs contain data in their code segment. Static disassemblers aim to understand the contents of code segments using two types of disassembly – linear sweep or recursive traversal. Linear sweep ensures high code coverage. However, it cannot distinguish between real code and data in the code segment.

To overcome the problem of data in code segments, another method of disassembly must be used. This method is recursive traversal, which only treats a region of the code segment as code if it can statically prove a control-flow path to it exists. static control flow paths are only known through direct CTIs. For indirect CTIs, the targets are not statically known and the target is only reachable via indirect CTIs.

A third limitation of static binary rewriting is that some benign programs contain obfuscated code, in which case static rewriting can break the program. The relevant kind of obfuscation is control-flow obfuscation whose goal is to mislead disassemblers so that they cannot reverse-engineer binaries.

B. Dynamic Binary Rewriters

There are two main types of dynamic binary rewriters: in-place designs, and code-cache based designs. We will go over them briefly.

In-place designs, such as BIRD [21] have lower overhead in comparison to code-cache based designs by avoiding the high overhead incurred by maintaining the code cache; however, they fail to support some of the features which may happen relatively frequently in benign binaries such as obfuscation, dynamically-generated and self-modifying code. The reason BIRD does not work for obfuscated code is that it assumes both the fall through and destination of a conditional branch are code, which may not be true in obfuscated code. Further, BIRD does not support self-modifying code. The reason is that once they disassemble code from a location, they never change the disassembly even if the code is overwritten.

 ISBN: 978-1-61208-752-8

Unlike static and in-place dynamic rewriters, **code-cache based** dynamic rewriters are robust and can correctly rewrite all programs. However, existing rewriters have high overhead that is generally unacceptable for deployment on live systems. Two of the most popular code-cache based dynamic rewriters are DynamoRIO [2] and Pin [11] with 1.2x and 1.54x runtime overhead, respectively, on average for the full SPEC'06 benchmark suite *even without any instrumentation inserted*. Dyninst's version 2011 [22] is another code-cache based design which has 1.2x overhead for the same benchmark. Vulcan [23] is another dynamic binary rewriter which has a very strong API for adding instrumentation; however, it has very high overhead, around 2x to 3x compared to uninstrumented binaries.

VIII. Conclusion

In this paper, we have developed a novel design for a fully optimized, low-overhead binary rewriter which is robust like other dynamic binary rewriters. Due to its low overhead, it is practical to be used in real-time systems. Our experiments show that the overhead of DynamoRIO is 1.16x, whereas the overhead of RL-Bin is 1.05x.

In our future work, we will develop an instrumentation API for RL-Bin that is going to be both efficient and flexible. We aim to have a similar set of APIs to existing tools [24], [25] so that users can adapt to RL-Bin with minimal effort. We are also exploring other adversarial and obfuscation techniques against binary rewriters and the methods to circumvent them. In the near future, we will release RL-Bin's binary for non-commercial uses, similar to how Pin [11] is licensed.

References

[1] D. Shackleford, "A new era in endpoint protection," https://go.crowdstrike.com/rs/281-OBQ-266/images/ReportSANSProductReview.pdf, 2017, retrieved: October, 2019.

[2] D. L. Bruening, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, Massachusetts Institute of Technology, 2004.

[3] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Securing untrusted code via compiler-agnostic binary rewriting," in Proceedings of the 28th Annual Computer Security Applications Conference. ACM, 2012, pp. 299–308.

[4] C. Zhang et al., "Practical control flow integrity and randomization for binary executables," in Security and Privacy (SP), 2013 IEEE Symposium on. IEEE, 2013, pp. 559–573.

[5] M. Olszewski, J. Cutler, and J. G. Steffan, "Judostm: A dynamic binary-rewriting approach to software transactional memory," in Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques. IEEE Computer Society, 2007, pp. 365–375.

[6] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in Proceedings of the 2012 ACM conference on Computer and communications security. ACM, 2012, pp. 157–168.

[7] A. Roy, S. Hand, and T. Harris, "Hybrid binary rewriting for memory access instrumentation," ACM SIGPLAN Notices, vol. 46, no. 7, 2011, pp. 227–238.

[8] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "Tainteraser: Protecting sensitive data leaks using application-level taint tracking," ACM SIGOPS Operating Systems Review, vol. 45, no. 1, 2011, pp. 142–154.

[9] "Binary obfuscation project tool," https://www.codeproject.com/Articles/856846/Binary-Obfuscation, retrieved: October, 2019.

[10] I. V. Popov, S. K. Debray, and G. R. Andrews, "Binary obfuscation using signals." in USENIX Security Symposium, 2007, pp. 275–290.

[11] C.-K. Luk et al., "Pin: building customized program analysis tools with dynamic instrumentation," in Acm sigplan notices, vol. 40. ACM, 2005, pp. 190–200.

[12] G. Ravipati, A. R. Bernat, N. Rosenblum, B. P. Miller, and J. K. Hollingsworth, "Toward the deconstruction of dyninst," Comput. Sci. Dept., Univ. Wisconsin, Madison, Tech. Rep., 2007.

[13] M. Wenzl, G. Merzdovnik, J. Ullrich, and E. Weippl, "From hack to elaborate technique—a survey on binary rewriting," ACM Computing Surveys (CSUR), vol. 52, no. 3, 2019, p. 49.

[14] A. Majlesi-Kupaei, D. Kim, K. Anand, K. ElWazeer, and R. Barua, "Rl-bin, robust low-overhead binary rewriter," in Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation. ACM, 2017, pp. 17–22.

[15] K. Anand et al., "A compiler-level intermediate representation based binary analysis and rewriting system," in Proceedings of the 8th ACM European Conference on Computer Systems. ACM, 2013, pp. 295–308.

[16] A. Eustace and A. Srivastava, "Atom: A flexible interface for building high performance program analysis tools," in Proceedings of the USENIX 1995 Technical Conference Proceedings. USENIX Association, 1995, pp. 25–25.

[17] B. Schwarz, S. Debray, G. Andrews, and M. Legendre, "Plto: A link-time optimizer for the intel ia-32 architecture," in Proc. 2001 Workshop on Binary Translation (WBT-2001). Citeseer, 2001.

[18] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere, "Diablo: a reliable, retargetable and extensible link-time rewriting framework," in Signal Processing and Information Technology, 2005. Proceedings of the Fifth IEEE International Symposium on. IEEE, 2005, pp. 7–12.

[19] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "Pebil: Efficient static binary instrumentation for linux," in Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on. IEEE, 2010, pp. 175–183.

[20] D. Kim et al., "Dynodet: Detecting dynamic obfuscation in malware," in International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, 2017, pp. 97–118.

[21] S. Nanda, W. Li, L.-C. Lam, and T.-c. Chiueh, "Bird: Binary interpretation using runtime disassembly," in Code Generation and Optimization, 2006. CGO 2006. International Symposium on. IEEE, 2006, pp. 12–pp.

[22] A. R. Bernat and B. P. Miller, "Anywhere, any-time binary instrumentation," in Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools. ACM, 2011, pp. 9–16.

[23] A. Edwards, H. Vo, and A. Srivastava, "Vulcan binary transformation in a distributed environment," Microsoft Research, Tech. Rep., 2001.

[24] "Intel pin api," https://software.intel.com/sites/landingpage/pintool/docs/81205/Pin/html/group__API__REF.html, retrieved: October, 2019.

[25] B. Buck and J. K. Hollingsworth, "An api for runtime code patching," The International Journal of High Performance Computing Applications, vol. 14, no. 4, 2000, pp. 317–329.

 ISBN: 978-1-61208-752-8

 ISBN: 978-1-61208-752-8