RL-Bin, Robust Low-overhead Binary Rewriter

Amir Majlesi-Kupaei
University of Maryland
majlesi@umd.edu

Danny Kim
University of Maryland
dannykim@terpmail.umd.edu

Kapil Anand
SecondWrite LLC
kapil.anand@secondwrite.com

Khaled ElWazeer
SecondWrite LLC
wazeer@secondwrite.com

Rajeev Barua
University of Maryland
barua@umd.edu

ABSTRACT

Binary rewriters are used to ensure security properties or optimize and reduce runtime of existing binary applications. Existing binary rewriters are either static or dynamic, and both have severe shortcomings. Existing static rewriters have low overhead, but cannot rewrite correctly for all binaries. Existing dynamic rewriters are robust, but have high overhead – for example, for a subset of SPEC'06 benchmarks we measured, their overhead is 1.59X for PIN and 1.32X for DynamoRIO. Because of this high overhead, they are limited to off line testing, and cannot be used in deployment.

We have built the first binary rewriter called RL-Bin which can rewrite all binaries correctly, but has low overhead (averaging 1.09X for our programs.) This makes it practical for continuous use in deployed software for the first time. This paper represents an early snapshot of on-going research, and we hope to bring this overhead down even further in the future. We have also shown how RL-Bin can be used to enforce CFI, a security mechanism.

CCS CONCEPTS

• **Security and privacy** → *Software security engineering*;

KEYWORDS

Binary Rewriting, Control Flow Integrity, Enforce Security Properties

ACM Reference Format:
Amir Majlesi-Kupaei, Danny Kim, Kapil Anand, Khaled ElWazeer, and Rajeev Barua. 2017. RL-Bin, Robust Low-overhead Binary Rewriter. In *Proceedings of FEAST'17*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3141235.3141245

1 INTRODUCTION

Ensuring security properties in programs may take two approaches. First, we would like to ensure that the application's vulnerabilities against malicious attack are minimized or eliminated. Second, in some cases we may also want to enforce security-related constraints on any running code, to prevent it from doing unauthorized actions, either because the developer made a mistake or is not fully trusted, or control was hijacked, and attacker code is running. These two approaches – reducing vulnerabilities and enforcing security constraints – are not mutually exclusive. Both may be adopted to improve security.

In both scenarios above, it is highly desirable for the computer system's user to ensure security at the level of binary code, instead of source code. Reasons include (i) the user of the code may not have access to the source code; or (ii) even when source code is available, it may come from a variety of sources, including legacy code and 3rd party code, not all of which may have followed good security practices, and moreover, the complete compiled binaries may not be hardened with security properties.

Although research efforts in binary rewriting have been conducted for more than a decade, a guaranteed-correct, low-overhead binary rewriter still does not exist. Much progress has been made, however. Static binary rewriting approaches have shown how many high-level artifacts can be recovered from binaries, but the approaches are not robust, in that they do not work for all binary programs. Dynamic binary rewriting approaches have shown how robustness can be achieved, but pay a high price in run-time overhead, making them largely impractical for real-world deployment use.

In this paper, we will show how we can design a dynamic rewriter that inherits the robustness of dynamic approaches, but selectively uses just-in-time code analysis (similar to static analysis, but at run-time) to reduce the overhead to very low levels. To appreciate how it works, next we overview previous approaches for static and dynamic binary rewriting.

1.1 Limitations of static binary rewriters

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.

Current Static rewriting approaches include [3, 8, 9, 11, 16, 19]. SecondWrite [3], aims to recover compilable source code from binaries, initially output as LLVM IR, which could then further be compiled into rewritten executable code. The Ida pro disassembler [9] generates C-like pseudo-code to aid human understanding of the binary code. The pseudo-code is not meant to be executed, and often would not work if it is attempted to be compiled. ATOM [8] provides a flexible interface for code instrumentation which helps in the development of program analysis tools. Diablo [19] aims to provide a framework for link-time program transformation with whole program optimization and instrumentation. Pebil [11] is another static binary rewriter focused on achieving efficient, low-overhead binary instrumentation. They have built very efficient instrumentation tools 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 runtime 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 execution of the program. Hence, these codes will not be observed and instrumented, leading to incomplete code coverage. An attacker can use just the vulnerability inside the dynamically generated code to take control of the whole program. If that is the case, all the instrumentations in other parts of the program would not help. Another problem is that even for the code that is statically visible, its context may include dynamically generated code, leading to incomplete characterization of its behavior.

Dynamically generated code is quite common in benign applications. In a study recently done in our research group, we observed that 29 out of 120 benign applications contain some dynamically generated code, which is usually used for supporting execution of user scripts. The 120 benign programs included well-known commercial third-party binaries commonly used by many users of computer systems. This means that all the implementations which use static binary rewriters would fail to guarantee the security of the system for 24 percent 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, but cannot distinguish between code and data, especially if the data is a valid sequence of instructions, which is usually the case for x86 instruction set architecture. Hence it is not safe to use for a binary rewriter, since linear sweep can mistake data to be code and rewriting it will break the program.

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 from the beginning of the program. Unfortunately statically control flow paths are only known through direct control transfer instructions (CTI) (i.e. those CTIs whose targets are constants.) For indirect CTIs whose targets are computed at run-time, the targets are not statically known. The result is that a lot of the code in the program that is only reachable via indirect calls cannot be proven to be code. This results in incomplete code coverage in static binary rewriting, leading to incomplete implementation of security properties.

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 hackers cannot reverse engineer binaries to understand them or steal their IP. One example of obfuscation is unconditional branches which are converted to conditional branches where one path is never taken and leads to data, but can confuse the disassembler into thinking it is code, rewriting which can break the program. Another example of obfuscation is where a programmer-registered exception handler can redirect control flow to a point in the program other than where the exception was triggered, leading to control-flow paths that cannot be statically discerned. When even a single instance of obfuscation (or code that appears to be obfuscation) is present, the static binary rewriter will likely break the program. In our experiments, we have found that about 1% of benign programs have obfuscation. The serious consequence is that these programs will not run correctly any longer after rewriting, which is unacceptable.

1.2 Limitations of existing Dynamic Binary rewriters

Unlike static rewriters, dynamic rewriters are robust and can correctly rewrite all programs. However existing dynamic rewriters have high overheads that are generally unacceptable for deployment on live systems. Two of the most popular dynamic rewriters are DynamoRio [5] and Pin [12] with 1.2x and 1.54x run-time overhead, respectively, on average for the full SPEC'06 benchmark suite *even without any instrumentation inserted*. Our interaction with industry personnel has revealed that Run-time overhead above a few percent is unacceptable in deployment. Dynamic binary rewriters copy all code that executes into another memory region called a *code cache*, in which the code can be instrumented and then executed from. The code cache is useful because it ensures robustness – if a piece of data is mistakenly assumed to be code and rewritten, the program still works, because the original copy of the code segment with data in it is still unchanged – only the code cache was changed.

The overhead of dynamic rewriters are caused by three factors. First, copying code into the code cache is expensive at run-time. Second, and more seriously, the target addresses of indirect CTIs 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 actually very common, mostly in the form of return instructions, which return control to caller functions from callees, as well as function pointer calls, and calls to virtual functions in object-oriented languages such as C++. This translation process is inevitable for dynamic binary rewriters, since the original destination address in the program is different from the address of the rewritten code inside the code cache. A third cause of the overhead of dynamic rewriters is the increased pressure on the instruction cache, because each code portion is copied to a code cache, and in a few cases, both copies may be cached.

2 RL-BIN: PROPOSED BINARY REWRITING APPROACH

We propose a new approach to binary rewriting that overcomes the limitations of both static binary rewriting, in they are not robust; and the limitations of existing dynamic binary rewriters, in that they incur high overhead. Our proposed binary rewriting tool is called **RL-Bin**, (Robust, Low overhead Binary Rewriting). Unlike static rewriters, it is robust, and works for all binary programs. Instead, it is a dynamic binary rewriter, that unlike existing dynamic binary rewriters, incurs low run-time overhead, and hence is practical for use in deployment.

We build on prior work on the most advanced static binary rewriter [17] [3] in existence today. From that work we can learn about the shortcomings of existing static rewriters, but we can also gain a lot of insights. This research borrows ideas from static rewriting, but employs them in a dynamic binary rewriter instead, inside of just-in-time code translation modules that can reason about the code, and reduce its overhead.

In this paper, we will show how RL-Bin works by designing, implementing, and evaluating the scheme. In the future, we will extend the early prototype we have right now, which encodes a basic not fully optimized design. RL-Bin rewriter works as follows. Its main approach is to avoid a code cache, and instead rewrite the binary in-place in its run-time memory image. Avoiding the code cache eliminates most of the factors leading to run-time overhead mentioned earlier, leading to a much faster binary rewriter. However, for correctness, rewriting in-place requires us to ensure two properties exist in our scheme. First, we are going to discover code from *all* the memory addresses that contain instructions that will be executed at some time during the execution of the program. Second, we need to ensure that we instrument a memory location only if we can ensure that location contains code and not data. To this end, we don't assume any location contains code unless some instruction is executed from that address.

2.1 Methodology

The first intuition behind RL-Bin is to add instrumentation at run-time that monitors discovery of new code. To generate the correct and safe rewriting, our method assume that a block of memory is code, only if we discover an actual control transfer during runtime. Therefore, we solve this problem by the insertion of instrumentation into the program at strategic locations where control flow may be transferred to more than one destination. Thereafter, if for any of these instrumentations, all possible successor code had already been discovered, the instrumentation can be removed, resulting in a low overhead scheme.

More specifically, here is how RL-Bin works. Whenever an application begins its execution, our binary rewriter intercepts it and take control of the program. It starts recursive traversal from entry point of the program, and stops at any CTI with more than one possible outcome. These include conditional direct branches and indirect CTIs, but not unconditional jumps. For those uncertain successor CTIs, we insert instrumentation just before the CTI to check whether the actual execution target is registered previously as code. If it has not, then that new code will be registered.

Using this approach, more and more code is discovered during runtime. This method would 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.

If at any point, both outcomes of a conditional branch are registered as code, then the instrumentation at that branch can be removed. In this way, in the steady state, most or all of the checks before direct conditional branches are removed and only the check before indirect CTIs are remained. Since the number of indirect CTIs is low compared to the number of CTIs with uncertain successors, the overhead is significantly reduced.

The overhead of the above scheme can be further reduced by removing checks, at the most common instances of indirect CTIs, which are return instructions. Intuitively, a check before a return instruction can be removed if we can prove that the return instruction always transfers control back to the instruction after the original call instruction which is the case for most functions. To remove the check for a certain function, we need to prove that the function is “safe” in that it cannot modify its own return address. We have designed certain just-in-time analysis algorithms, based on static analysis concepts, by which the safety of vast majority of functions can be established before their execution. For such safe functions, the instrumentation before the return instruction can be removed because it is not needed.

Our scheme avoids the high overhead of code cache-based dynamic binary rewriters. The first source of overhead was copying the code to the code cache which is obviously not present in our method. The second and the most significant source of overhead, is dynamic address translation of indirect CTIs. This address translation is inevitable in the code cache-based rewriters, since the actual address of destination resides in the program original memory space and it must be be translated to the corresponding address in the code cache. However, in our scheme, because code is not moved, we do not actually have to translate this address. The only check needed in our scheme, is to ensure that the destination of the indirect CTI is valid. In many cases, if we can ensure that the indirect CTI is actually going to transfer to a known valid destination just-in-time before the block's execution at run-time, then we will be able to avoid instrumenting that indirect CTI at all, and that would lead to significant reductions of overhead. The third source of overhead (instruction cache pressure) would also be lower because unlike a code cache, in-place rewriting does not keep two copies of code.

We will also build additional algorithms for safe handling of user registered exceptions, to ensure control flow is always visible to the face of our binary rewriter.

Once correct and low-overhead binary rewriting is ensured using the approach, in this ongoing work we will use RL-Bin to analyze the code, and then insert code instrumentation into the application, where the instrumentation ensures the security policies in question.

3 IMPLEMENTATION AND RESULTS

We have completed and tested an early prototype of the above method. Most of the code is written in C++ programming language, while there are some functions which are written in x86 assembly, for the sake of optimization. Our Target architecture is x86 and we decided to choose windows operating system, since most of security exploits target windows binaries.

For our experimental setup, we used a subset of SPECint'06 benchmark and used its reference data set. In comparison with SPECfp'06 applications, reducing overhead of integer benchmarks

Application	PIN	DynamoRIO	RL-Bin Unoptimized	RL-Bin Semi-optimized	RL-Bin Fully optimized
perlbench	1.89x	1.73x	7.91x	1.38x	1.21x
bzip2	1.09x	1.05x	4.59x	1.06x	1.04x
gcc	2.35x	1.47x	6.42x	1.29x	1.12x
mcf	1.02x	1.01x	1.62x	1.02x	1.01x
Average	1.59x	1.32x	5.14x	1.19x	1.09x

Table 1: Normalized run-time of rewriters without added instrumentation. . A run-time of 1.0 is the run-time of the original unmodified program without rewriting. For example, if the overhead of the rewriter is shown as 1.89X, that means the overhead of the rewriter adds is 89% without added instrumentation.

is more challenging due to higher frequency of indirect control transfer instructions.

Our experiments are done on a single core of Intel Core i5, 2.3GHz CPU with 3 Mb cache and 8 Gb DDR3 memory on 32 bit Windows 7 operating system.

The goal of our binary rewriter is to perform light instrumentation efficiently and it is not optimized for heavy instrumentations such as basic block counting which will significantly modify the binary image of the application. As a result, performance overhead of applications running under binary rewriter without instrumentation has been measured.

The results are shown in table 1. Performance overhead was measured for PIN and DynamoRIO as well as unoptimized, semi-optimized, and fully optimized versions of RL-Bin.

The basic unoptimized version instruments every single CTI in the code and does not remove the instrumentation when their existence is no longer necessary.

The semi-optimized version has many improvements such as removing instrumentations before all unconditional and some conditional direct branches when allowed. Also, instrumentations are improved and written in assembly and optimized as much as possible. Another optimization is function cloning of leaf functions which removes the instrumentation of return instruction of these functions.

Fully optimized RL-Bin uses static analysis during runtime to detect return instructions, the most common form of indirect CTIs, which do not need to be instrumented. In fact, we can show that these instructions will return to one of the several memory locations which have been seen and analyzed by our binary rewriter.

The high overhead of *perlbench* and *gcc* are due to the high number of dynamically executed indirect CTIs. As it can be seen, the main source of overhead for every dynamic rewriter is the presence of such instructions which cannot be managed prior to execution. Our overhead will go further down by refining our last optimization technique to eliminate the instrumentation for more return instructions.

Currently, it can be seen that we outperform PIN and DynamoRIO by a huge margin. In fact, the overhead of PIN and DynamoRio is 1.59X and 1.32X respectively, whereas the overhead of RL-Bin is 1.09X (9%). In future work, we hope to lower the overhead of RL-Bin further to low single digits, making it the first robust rewriter suitable for use in deployed software.

4 RL-BIN APPLICATIONS

4.1 CFI

One way to ensure benign applications are not hijacked during attacks is to use a defense mechanism called Control Flow Integrity (CFI). CFI is one of the most effective application control-flow hijack defense methods invented to date, and has many nice theoretical properties ensuring its soundness and scope of defense. Here is how CFI works. First, the control flow graph (CFG) of the application is calculated using source code analysis, binary analysis, or execution profiling. Then CFI ensures that software execution follows one of the paths in its intended CFG. In order to enforce this security policy, runtime checks are instrumented before control transfer instructions to make sure that the control-transfer instruction (CTI) is actually taking one of the edges from the CFG. The target address must be the destination of one of the outgoing edges from the current node. These runtime checks prevent any unintended control flow transfers during the program's execution.

CFI can protect against a variety of attacks that are based on hijacking the control-flow of a benign application. These include stack-based buffer overflow attacks, heap-based jump-to-libc attacks, and return oriented programming (ROP). In any of these attacks, the attacker needs to transfer control to the payload code which could be injected by the attacker or may already be resident on the computer. During this step, CFI intercepts the CTI, checks its destination against allowed destinations, and thus terminates any attack before execution of any malicious code.

Abadi et al [2] have derived strong theoretical properties of CFI. CFI ensures that every execution step of an instrumented program is either an attack step in which the program counter does not change, or a normal step to a state with a valid successor program counter. Thus, despite attack steps, the program counter always follows the CFG. This would imply that the attacker cannot cause the execution of code that is unreachable in the CFG.

4.1.1 CFI Related Work. The Control Flow Integrity scheme was first introduced in 2005 by Abadi et al[1]. Its goal is to monitor all CTIs to make sure that the application is following one of the edges in its CFG, which is determined in advance. This would ensure that the program is behaving according to its intended control flow. In the paper, their security policy dictates that a return instruction must transfer the control to the next instruction after the call site of the function. Also, indirect calls, may go to any function whose address is taken. These functions will be discovered by a flow-insensitive analysis of relocation entries in the binary. Then a unique ID is assigned to the destination of each indirect CTI and then CTIs will be instrumented to check the unique ID of the destination against the ID which is determined ahead of time. If the two ID match with each other, it means that the path existed in the precomputed CFG. Otherwise, the program detects the ID mismatch and reports the error. Their instrumentations were added using Vulcan [7] which is a static binary rewriter. The overhead caused by added instrumentation was 16 percent on average for SPEC benchmark, which is fairly high for deployment on the live systems.

Modular CFI (MCFI)[13] is another low overhead techniques proposed for enforcing CIF which supports separate compilation of modules. While MCFI has acceptable overhead (5%), there are two reasons why our proposed method is more effective. First, MCFI is dependent on having access to source code, which is not available for most third party binary applications, while our method uses only the binary file. Secondly, MCFI instrument the code statically, which makes their method ineffective for the dynamically generated or self-modifying code.

Some other CFI implementations have been proposed with just a few percent overhead [20]. However, these implementations require source code and the modification would be done as part of the compiling process. Other related works, have tried to optimize the checks and succeeded to decrease the overhead to just about 3.6 - 8.6 percent [21]. However, they still rely on static binary analysis and that can lead to robustness problems, as discussed in Section 1. Using a dynamic binary rewriter to perform the instrumentations for CFI has been tested [15] and the overhead was reported to be around 20 percent which is mostly due to the high overhead caused by the binary rewriter itself. This is still too high for use in deployed software.

4.1.2 Implementation of CFI using RL-Bin.

Traditionally, there are two main steps involved in the implementation of CFI. In the first step, the CFG of the program is calculated using static analysis. Original CFI [1] relies on relocation entries and other static information derived from static analysis of binary. This is problematic because the goal is to handle dynamically generated, self modifying, and obfuscated code (all of which are present in benign applications), for which the static information could be misleading. So, the goal is to stay away from static analysis. To accomplish this goal, we therefore need to adapt CFI to be purely dynamic. We no longer use pre-calculated CFG of the binary. Instead, our dynamically inserted checks will enforce the same security policies based on information obtained from dynamic execution of the program. In this paper we present a design of how to implement CFI in RL-Bin. Implementation is ongoing, and evaluation is future work.

In the second phase of CFI, dynamic checks must be inserted to ensure that only the paths in the CFG are followed. We use the instrumentation module of our planned RL-Bin binary rewriter to put these checks before indirect CTIs. Enforcing CFI based on purely dynamic information is possible because of the following reasons. First, CFI checks on returns can be done by instrumenting return instructions to check their destination to be the next instruction after one of the original function call sites which has been discovered so far. Also, returning to the most recent call site can be enforced by implementing a low overhead shadow stack. Similarly, CFI checks for indirect calls can also be implemented purely dynamically by checking the destination of the call to be one of the addresses seen so far using binary characterization [18], a method that looks for address constants in binaries to discover possible legal targets of indirect CTIs. In this way, a purely dynamic scheme for CFI can be implemented.

Since we have developed the binary rewriter, we have a significant advantage in comparison to just using a third-party rewriter. Instead of inserting new checks for CFI, some of the CFI checks could be integrated with RL-Bin's inherent run-time checks needed for its proper execution, and their optimizations can be integrated too. The integration of checks will significantly decrease the extra overhead from CFI. As a result, our implementation of CFI would have comparable overhead with those using static rewriters despite our implementation is being more robust.

4.2 Other security policies

CFI is by no means the only security policy that can be forced by RL-Bin. As a matter of fact, our rewriter can be used to instrument any security policy that can be implemented with light instrumentation. The end result will be a low overhead tool which will be practical for deployment on end-user systems. Examples of such security policies are stack-canary insertion [6] for return address modification detection, Program Shepherding [10], instruction-set randomization [4], and return address protection using shadow stacks [14].

5 CONCLUSION

In this paper, we proposed a new design for developing a binary rewriter which will lead to a tool which is as robust as other dynamic binary rewriters while the overhead is tolerable for monitoring real-time systems in practice. Our experiments show that the overhead of PIN and DynamoRio is 1.59X and 1.32X respectively, whereas the overhead of RL-Bin is 1.09X (9%).

For future work, we will examine several possible applications for RL-Bin including enforcing CFI. Using the proposed approach will lead to such low overhead that it would be practical to enforce CFI for every single application running on a live system.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 340–353.

[2] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. A theory of secure control flow. In *International Conference on Formal Engineering Methods*. Springer, 111–124.

[3] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. 2013. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 295–308.

[4] Stephen W Boyd, Gaurav S Kc, Michael E Locasto, Angelos D Keromytis, and Vassilis Prevelakis. 2010. On the general applicability of instruction-set randomization. *IEEE Transactions on Dependable and Secure Computing* 7, 3 (2010), 255–270.

[5] Derek L Bruening. 2004. *Efficient, transparent, and comprehensive runtime code manipulation*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[6] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks.. In *Usenix Security*, Vol. 98. 63–78.

[7] Andrew Edwards, Hoi Vo, and Amitabh Srivastava. 2001. Vulcan binary transformation in a distributed environment. (2001).

[8] Alan Eustace and Amitabh Srivastava. 1995. ATOM: A flexible interface for building high performance program analysis tools. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*. USENIX Association, 25–25.

[9] SA Hex-Rays. 2008. IDA pro disassembler. (2008).

[10] Vladimir Kiriansky, Derek Bruening, Saman P Amarasinghe, et al. 2002. Secure Execution via Program Shepherding.. In *USENIX Security Symposium*, Vol. 92. 84.

[11] Michael A Laurenzano, Mustafa M Tikir, Laura Carrington, and Allan Snavely. 2010. Pebil: Efficient static binary instrumentation for linux. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 175–183.

[12] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[13] Ben Niu and Gang Tan. 2014. Modular control-flow integrity. *ACM SIGPLAN Notices* 49, 6 (2014), 577–587.

[14] Pádraig OâĂŹSullivan, Kapil Anand, Aparna Kotha, Matthew Smithson, Rajeev Barua, and Angelos D Keromytis. 2011. Retrofitting security in cots software with binary rewriting. In *IFIP International Information Security Conference*. Springer, 154–172.

[15] Mathias Payer, Antonio Barresi, and Thomas R Gross. 2015. Fine-grained control-flow integrity through binary hardening. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 144–164.

[16] Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. 2001. Plto: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001)*. Citeseer.

[17] Matthew Smithson, Kapil Anand, Aparna Kotha, Khaled Elwazeer, Nathan Giles, and Rajeev Barua. 2010. Binary rewriting without relocation information. *University of Maryland, Tech. Rep* (2010).

[18] Matthew Smithson, Khaled ElWazeer, Kapil Anand, Aparna Kotha, and Rajeev Barua. 2013. Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE, 52–61.

[19] Ludo Van Put, Dominique Chanet, Bruno De Bus, Bjorn De Sutter, and Koen De Bosschere. 2005. 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, 7–12.

[20] Zhi Wang and Xuxian Jiang. 2010. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 380–395.

[21] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 559–573.