# Retrofitting Security in COTS Software with Binary Rewriting

Pádraig O'Sullivan [1], Kapil Anand [1], Aparna Kotha [1], Matthew Smithson [1], Rajeev Barua [1], and Angelos D. Keromytis [2]

[1] Electrical and Computer Engineering Department, University of Maryland
[2] Department of Computer Science, Columbia University

We present a practical tool for inserting security features against low-level software attacks into third-party, proprietary or otherwise binary-only software. We are motivated by the inability of software users to select and use low-overhead protection schemes when source code is unavailable to them, by the lack of information as to what (if any) security mechanisms software producers have used in their toolchains, and the high overhead and inaccuracy of solutions that treat software as a black box.

Our approach is based on *SecondWrite*, an advanced binary rewriter that operates without need for debugging information or other assist. Using SecondWrite, we insert a variety of defenses into program binaries. Although the defenses are generally well known, they have not generally been used together because they are implemented by different (non-integrated) tools. We are also the first to demonstrate the use of such mechanisms in the absence of source code availability. We experimentally evaluate the effectiveness and performance impact of our approach. We show that it stops all variants of low-level software attacks at a very low performance overhead, without impacting original program functionality.

## 1 Introduction

Despite considerable research and work on programmer education and tools, programming language and compiler support for security, hardware and operating system features, low-level software vulnerabilities remain an important source of compromises and a perennial threat to system security. While other sources of vulnerability have emerged more recently, such as SQL injection, cross-site scripting (XSS) and cross-site request forgery (XSRF), binary-level vulnerabilities continue to be discovered in very popular software and to be exploited for fun and profit [12].

The lack of convergence to a comprehensive solution can be attributed to several factors, consisting of a mix of the technical and non-technical. At the core, there exists a fundamental dichotomy in the capabilities and motivation of producers and consumers of software, vendors and end-users/administrators respectively. On the one hand, software producers are probably in the best position to both proactively and reactively prevent and mitigate such vulnerabilities: they have access to the source code, the compiler tool chain, and the developers themselves. As a result, they can apply security mechanisms that offer high coverage and effectiveness at low overhead, because they are applied at the point where the most semantic knowledge about the program and the code is available. On the other hand, it is software consumers that face the risk and bear the costs of compromise due to software vulnerabilities and are the most motivated to take action, often localized, to mitigate a newly discovered vulnerability. However, consumers often only have access to the program binary and configuration files. Thus, absent vendor patches (which can often take a long time and may contain bugs [32]) consumers can only use security mechanisms that treat the software as a black box. Inevitably, such mechanisms resort to isolation (*e.g.,* through a virtual machine) or to behavioral detection (*e.g.,* system call monitoring), with attendant costs, complexity and risk. Even security-conscious software consumers often cannot properly evaluate the risks they face because they do not know what security mechanisms, if any, a producer has used in their development process and tool-chain [22].

We present a new mechanism based on advanced binary rewriting that seeks to bridge the gap between incentive/motivation and capabilities on the consumer side. Our approach allows end users to retrofit powerful security mechanisms into third-party, binary-only software. These mechanisms are well-known, and some of them have been *partially* integrated in separate tools and development environments (*e.g.,* ProPolice in *gcc* and the optional */GS* flag in Visual Studio). Our system allows end-users to ensure that the software they run on their systems uses any and all such features, regardless of the choices or capabilities of vendors[3]. Furthermore, our approach allows end-users to selectively apply different defense mechanisms to different parts of the program, based on their own analysis, risk assessment, and knowledge of potential or actual vulnerabilities in the code. Essentially, we provide the nearly the same self-defense capabilities that open-source software users *can* utilize to users of binary-only software[4].

---

[3] Not all development tool-chains support a given security feature, while vendors and products are often intimately tied to them. As a result, there is considerable reluctance by vendors to switch to a "better" compiler, for example, even if such existed.

The contributions of this paper are twofold. First, we present a powerful binary-rewriting framework in the context of software security. Specifically, we investigate the ability of such a system to retrofit known invasive, powerful and low-overhead security mechanisms to program binaries, in the absence of source code or even debugging symbols. Second, we evaluate the effectiveness and efficiency of our scheme and of the retrofitted security mechanisms, as compared to other ways in which these and similar security mechanisms can be applied to software. We conclude that a system such as ours would enable software consumers to protect themselves at the same level of effectiveness as if vendors had taken similar steps (*i.e.,* used the same security techniques) and at equally low overhead. Thus, we believe that we have removed a significant factor in improving the overall security posture of systems against low-level software compromises.

An additional contribution of this paper is that we have carefully chosen a set of complementary and effective schemes that, taken together, achieve the goal of defending against all types of buffer overflow attacks at the lowest combined run-time cost. The totality of our schemes protect against buffers on the global, stack, and heap segments from overflowing onto a variety of (usually code) pointer locations that are vulnerable to attack, including return addresses, function pointers, indirect branch pointers, *longjmp* buffers, and base pointers. This is an important practical contribution in itself, as this is the first solution in the literature to retrofit a comprehensive set of protections against buffer overflow attacks, which are still very common, into arbitrary new and legacy binaries. We intend to make this tool available publicly soon.

The remainder of this paper is organized as follows. Section 2 gives an overview of related work in binary rewriting and binary-only software security mechanisms. Section 3 presents background on binary rewriting, and how rewriting relates to security. We describe the methods we have chosen in Section 5, and discuss experimental results in Section 6. We conclude with our thoughts for future work in Section 7.

## 2 Related Work

Our work is related to many techniques that attempt to defend against attacks on vulnerabilities in applications. In this section, we elaborate on some of the pieces of work most closely related to ours. We present the various attack techniques utilized by attackers that are relevant for this research and then we go on to present various techniques proposed for mitigating these attack techniques. We also briefly discuss related work in binary rewriting.

### 2.1 Catalog of Attack Techniques

**Buffer Overflow Attacks** A buffer overflow refers to a situation that can occur when code writes into a bounded array, or buffer, and the writes are not correctly guarded against overflow. Data copied into the buffer whose length is larger than the buffer's size is referred to as a buffer overflow. Writes into a buffer that are not correctly guarded may overwrite and corrupt a variety of vulnerable locations that may also be stored nearby the buffer, including return addresses, base pointers, function pointers, and *longjmp* buffers. Although buffer overflows have historically most often occurred on the stack, they are also possible on heap and global segment buffers. For example a global buffer's overflow may overwrite a function pointer or *longjmp* buffer also in the global segment.

Buffer overflow attacks work by changing the value of the code pointer stored in vulnerable locations such as return addresses, function pointers and *longjmp* buffers. The code pointer is overwritten by a new value pointing to code of the attacker's choice. Base pointers are also vulnerable even though they are not code since they can be used for attacks [31]. A return address attack was first described in detail by AlephOne in 1996 [1]. However, attacks of this kind date back to before 1988 when the technique was used in the *fingerd* exploit of the Morris worm.

Commonly, an attacker would choose their input data so that the machine code for an attack payload would be present at the modified return address. When the vulnerable function returns, and execution of the attack payload begins, the attacker has gained control of the behavior of the target software. The attack payload is often called shellcode, since a common goal of an attacker is to launch a command line interpreter (referred to as a shell in UNIX like environments) under their control.

**Return-to-libc Attacks** As an alternative to supplying executable code (referred to as direct code injection), an attacker might be able to craft an attack that executes existing machine code (indirect code injection). This class of attacks has been referred to as jump-to-libc or return-to-libc (arc injection [9] has also been used to refer to this class of attacks) because the attack often involves directing execution towards machine code in the standard C library (libc) [9]. The standard C library is often the target for attacks of this type since it is loaded in nearly every UNIX program and it contains routines of the sort that are useful for an attacker. This technique was first suggested by Solar Designer in 1997 [27]. Attacks of this kind can evade defense mechanisms that protect the stack such as stack canaries and it is also effective against defenses that only allow memory to be writable or executable.

---

[4] Just because open-source software users *can*, does not mean they generally *do* assess or modify/secure their installations.

Traditionally, attacks of this kind have targeted the system function in the standard system library which allows the execution of an arbitrary command with arguments. In this case, the return address of a vulnerable function would be modified to point to the address of the system function which would then be executed with attacker supplied arguments. Since the system function executes a command on a system, if an attacker can control the arguments to this function, they could execute an arbitrary command on the system under attack. However, recent attacks have been demonstrated which do not depend on calling functions in the standard C library.

## 2.2 Catalog of Defense Techniques

**Compile Time Defenses** StackGuard [8] places a 'canary' (a memory location) on the stack between local variables and the return address. This canary value is designed to warn of stack corruption since validating the integrity of the canary value is an effective means of ensuring that the function return address has not been corrupted. Microsoft's compiler also supports the insertion of stack canaries with the /GS option.

ProPolice [10] is similar to StackGuard in that it places a canary value on the stack. However, ProPolice also changes the stack layout to place arrays and other function-local buffers above all other function-local variables. Copies of all function arguments are also made into new, function-local variables that also sit below any buffers in the function. As a result, these variables and arguments are not subject to corruption through an overflow of these buffers.

PointGuard [7] protects all code pointers within a program. The defense consists of encrypting pointer values in memory and only decrypting the pointers when they are loaded into CPU registers. The encryption key used is a randomly generated during process creation and is thus unknown to an attacker. Without knowledge of the encryption key, an attacker can not modify any value stored in memory. As a result, pointer values are not subject to corruption.

StackGuard, PointGuard, and ProPolice involve compile-time analysis and transformation. Thus, unless the source code for an application is available, these techniques can not be used thereby hindering the ability to easily deploy these techniques. In practice only the developer can use these defenses, and only if the compiler his or her organization uses supports it. Our techniques do not suffer from this drawback since they can be easily deployed on any binary produced from any source language and compiler, by not only the developer, but the end-user as well.

**Instruction Set Randomization** Instruction-set randomization [5] is a technique for protecting against buffer overflows (and many kinds of code injection attacks). This approach randomizes the underlying system's instructions so that foreign code injected by an attacker would fail to execute correctly since the attacker does not know the instruction set of the target system. However, as mentioned by the authors in [5], the main drawback of this technique as applied to binary code that it needs specialized hardware support in the processor. Thus, even though instruction-set randomization offers a strong defense against buffer overflow attacks the fact that unless it is supported by specialized hardware, it incurs significant overheads means that it is unlikely to see adoption in practice for the foreseeable future.

Strata (a dynamic binary translation framework) and Diablo (a link-time binary rewriter) were used to implement instruction set randomization [16]. Diablo is used to prepare a binary for string encryption and introduce the information necessary to detect foreign code. Strata is then used to provide the necessary virtual execution environment for safe execution. The main contribution of this work is that the instruction-set randomization implementation is efficient while requiring no special hardware support. However, the runtime overheads reported are still high because of the necessary software ISA translation at run-time, and the inherent overheads of a dynamic translator. These run-time overheads and likely to limit the practical adoption of such a system. Moreover any user of a dynamic binary rewriter must install it in addition to the application desired, making it inconvenient to use.

The static (off-line) binary rewriter we use suffers from none of these issues. No special hardware is required to utilize a binary rewriter and overheads are relatively low since no ISA translation is done, and since no dynamic translator is used, no additional software in addition to the application is needed for execution. In our system, if an original binary was compiled without optimizations, we often see a significant run-time improvement when rewritten.

**Address Space Layout Randomization** Address Space Layout Randomization (ASLR) can be seen as a relatively coarse-grained form of software diversity. ASLR shuffles, or randomizes, the layout of software in the memory address space. The common implementation of this scheme is at the OS level. Thus, when a process is launched the address space layout of the process will be different from a previous invocation of the same process. It is effective at preventing remote attackers that have no existing means of running code on a target system from crafting attacks that depend on addresses. ASLR is not intended to defend against attackers that are able to control the execution of a piece of software; it is mainly intended to hamper remote attackers from attempting to use the same attack repeatedly. Finally, its utility on 32-bit architectures is limited by the number of bits available for address randomization [25].

A binary rewriter could easily be used to provide a similar defense mechanism as ASLR. An interesting future avenue of research is to investigate software diversity through binary rewriting.

**Control Flow Integrity** Control Flow Integrity (CFI) [3] is a basic safety property that can prevent attacks from arbitrarily controlling program behavior. CFI dictates that software execution must follow a path of a control-flow graph that is determined ahead of time by analysis (in this case, static binary analysis is performed). CFI is enforced using static verification and binary rewriting (with Microsoft's Vulcan [28] tool) that instruments software with runtime checks. These checks aim to ensure that control flow remains within a given control-flow-graph. CFI is a very effective defense against buffer overflow attacks (and any attack which attempts to change a program's control flow) since any attempt by an attacker to divert the control flow of a program will be caught by CFI. However, the main barrier to CFI's adoption seems to be the overhead associated with the scheme. The average overhead of CFI in the prototype implementation is 16% on the SPEC2000 benchmarks. Also, unlike SecondWrite, the binary rewriter used by CFI depends on a binary being compiled with debug information which is usually not available in production binaries. If a binary is not compiled with debug information then CFI cannot be currently applied.

Our schemes implemented through our binary rewriter can provide the same level of protection as CFI. An additional advantage of our scheme is that our binary rewriter does not require access to any special information in an input binary unlike all previous binary rewriters (including the binary rewriter used in CFI) which require access to relocation or debug information.

**Program Shepherding** Program Shepherding [17] employs an efficient dynamic software machine-code interpreter (DynamoRIO [6]) for implementing a security enforcement mechanism. A broad class of security policies can be implemented using a machine interpreter such as DynamoRIO. For example, DynamoRIO could be used to enforce control-flow integrity. Program shepherding enforces a similar policy that imposes certain runtime restrictions on control flow such that an attacker can not alter a program's flow of control.

Program Shepherding can experience significant memory and runtime overheads, particularly on the Windows platform. The scheme requires an application and interpreter to be run simultaneously. The high overheads of interpretation in some cases are likely to limit adoption of Program Shepherding. Further, unlike using off-line rewriters like SecondWrite, Program Shepherding requires the installation of an extra piece of heavyweight software (DynamoRIO) in addition to the application to be run.

### 2.3 Related Work in Binary Rewriting

Binary rewriting and link time optimizers have been considered by a number of researchers. Binary rewriting research is being carried out in two directions: static rewriting and dynamic rewriting. Dynamic binary rewriters rewrite the binary during its execution. Examples are PIN [19], BIRD [20], DynInst [13], DynamoRIO [6], Valgrind [21], and the translation phase of VMWare [2]. Dynamic rewriters are hobbled since they do not have enough time to perform complex compiler transformations; they have been primarily used for code instrumentation and simple security checks in the past. Moreover dynamic rewriters do not have the time to perform deep code analysis needed to discover program features needed for static optimization of security checks. Finally dynamic rewriters encounter run-time overhead from the act of rewriting, which can be substantial. Given these drawbacks, we do not discuss dynamic rewriters further.

The methods in this research are primarily directed at static binary rewriters such as our rewriter, SecondWrite. Existing static binary rewriters include Etch [23], ATOM [11], PLTO [24], Diablo [29], and Vulcan [28]. Three points of novelty for our work are as follows. First, we are not aware of any rewriter adding our particular set of existing compile-time security schemes to binaries. Second, none of the existing rewriters employ a compiler level intermediate representation; rather they define their own low-level machine-code-like custom intermediate representation. This has several downsides: $(i)$ most existing rewriters cannot modify the stack layout since they do not distinguish individual objects on the stack. Hence they cannot implement security schemes that modify the stack; and $(ii)$ most existing rewriters recognize functions, but not their arguments or return values, and hence cannot deploy security schemes that employ these schemes. SecondWrite overcomes both these problems as we will describe in section 4.

A third point of novelty of our work is that all existing rewriters can only rewrite binaries that contain relocation or debug information. This information, present at link-time, is usually discarded in COTS binaries for two reasons – it is not needed for execution; and vendors legitimately fear it can be used to reverse engineer their binaries. Indeed of twenty commercial and open-source binaries we surveyed, *none contained either relocation or debug information*. As a result, existing binary rewriters would not be able to rewrite those binaries at all. In effect, existing binary rewriters can only be deployed by developers, not end-users. In contrast our rewriter (SecondWrite) can rewrite arbitrary binaries even without relocation or debug information, as we will describe in section 4. This renders our platform a uniquely powerful tool for allowing anyone to rewrite binaries from any source to enable any security scheme they want.

## 3 Background on binary rewriting

This section presents some background on binary rewriting and discusses how security enforcement interacts with it. Our approach relies on innovative binary rewriting schemes [4,26] incorporated into our binary rewriting infrastructure called SecondWrite. Binary rewriters are pieces of software that accept a binary executable program as input, and produce an improved executable as output. The output executable typically has the same functionality as the input, but is improved in one or more metrics, such as run-time, energy use, memory use, security or reliability.

**Advantages of binary rewriting** In recognition of its potential, binary rewriting has seen much active research over the last decade. The reason for great interest in this area is that binary rewriting offers additional advantages over compiler-produced optimized binaries:

- **Ability to do inter-procedural optimization.** Although compilers in theory can do whole-program optimizations, the reality is that they do little if any. Many commercial compilers - even highly optimizing ones - limit themselves to separate compilation, where each file (and sometimes each function) is compiled in isolation. In contrast, binary rewriters have access to the complete application all at once, including libraries. This allows them to perform aggressive whole-program optimizations to exceed the performance of even optimized code. This ability can be useful for security schemes as well; in particular for those schemes that rely on whole-program information such as call graphs and inter-procedural properties to either work at all, or to optimize fully.
- **Ability to do optimizations missed by the compiler.** Some binaries, especially legacy binaries or those compiled with inferior older compilers, often miss certain optimizations. Binary rewriters can perform these optimizations missed by the compiler while preserving the optimizations the compiler did perform. This property may help the rewriter overcome some of the overheads of security enforcement by improvements in program run-time.
- **Increased economic feasibility.** It is cheaper to implement a code transformation once for an instruction set in a binary rewriter, rather than repeatedly for each compiler for the instruction set. For example, the ARM instruction set has over 30 compilers available for it, and the x86 has a similarly large number of compilers from different vendors and for different source languages. The high expense of repeated compiler implementation often cannot be supported by a small fraction of the demand. This implement-once property is useful for security schemes as well.
- **Portable to any source language and any compiler.** A binary rewriter works for code produced from any source language by any compiler. This is a significant advantage for a security scheme such as the one presented in this paper. A scheme would not need to be ported to various compilers but would instead only need to be implemented once within a binary rewriter. Portability of rewriters aids security schemes implemented in them as well.
- **Works for hand-coded assembly routines.** Code transformations cannot be applied by a compiler to hand-coded assembly routines, since they are never compiled. In contrast, a binary rewriter can transform such routines. Applying security in a binary rewriter has the advantage of working for hand-coded assembly versus compiler implementation of security, which does not.

**Architecture of Binary Rewriter** The binary rewriter developed by our group and utilized for this research is named SecondWrite. Figure 1 presents an overview of the SecondWrite system. SecondWrite's custom binary reader and de-compiler modules translate the input x86 binary into the intermediate representation (IR) of the LLVM compiler. LLVM is a well-known open-source compiler [18] developed at the University of Illinois, and is now maintained by Apple Inc. LLVM IR is language- and machine-independent. Thereafter the LLVM IR produced is optimized using LLVM's pre-existing optimizations, as well as our enhancements, including security enforcement in this paper. Finally, the LLVM IR is code generated to output x86 code using LLVM's existing x86 code generator.

The front-end module consists of a disassembler and a custom binary reader which processes the individual instructions and generates an initial LLVM IR. This module reads the format of instructions from Instruction Set Architecture (ISA) XML files for the ISA in question, allowing for targeting of the rewriter to different ISAs. Currently SecondWrite rewrites x86 and ARM binaries. To give an idea of the effort needed for retargeting, consider that the sizes of

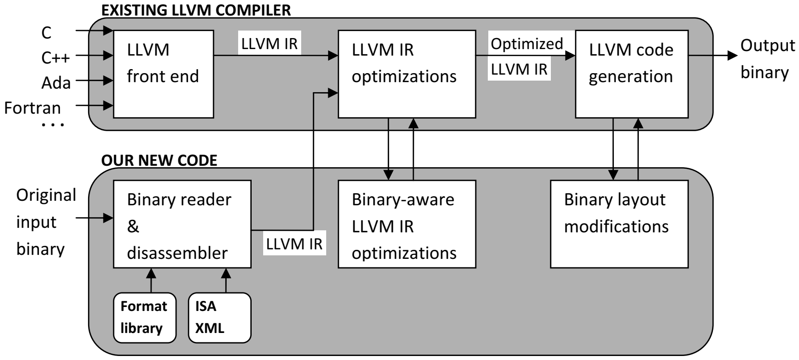

**Fig. 1.** SecondWrite system

the x86 and ARM XMLs are approximately 14000 and 1500 lines of code (LOC), respectively. The XML for x86 is much larger since it is a complex CISC ISA whereas ARM is RISC. This is a relatively small portion of the total size of SecondWrite, which exceeds 120,000 LOC (mostly C++). From this we can see the effort required for retargeting to a new RISC ISA is relatively modest (1-2 person-months in our estimate).

## 4 Innovations in SecondWrite

SecondWrite has three innovations that make it especially powerful, and a good platform for security enforcement. To be practical for security enforcement, a rewriter must satisfy three requirements. First, it must be able to rewrite stripped binaries (*i.e.*, those without relocation information) since most real-world binaries are stripped. Second, it must be able to rewrite the entire code, not just discoverable parts of it, thus achieving 100% code coverage. Third, it should rewrite the code to high-level IR, since some security schemes rely on high-level constructs such as functions, arguments, return values, and symbols. Below we describe why existing static rewriters do not provide any of these three capabilities, but SecondWrite does. We note that SecondWrite (and any similar tool) does not work with software that is either self-modifying or performs integrity self-checks.

**Rewriting without relocation information** A key innovation in SecondWrite is that it can rewrite stripped binaries, *i.e.*, those without relocation or symbolic information, unlike existing rewriters such as ATOM [11], PLTO [24], Diablo [29], and Vulcan [28] which cannot. Relocation information is generated by the compiler to help the linker in resolving addresses that can change when files are linked. Symbolic information may be inserted for debugging. However, production binaries almost never contain such information since linkers delete relocation information by default. The programmer may instruct the linker to retain such information. However corporations almost never release binaries with relocation and symbolic information since they are unnecessary for execution, and they fear such information can be used to reverse-engineer information about their code.

The requirement for relocation information in existing rewriters arises from the need to update the target addresses of control-transfer instructions (CTIs) such as branches and calls. When rewriting binaries, code may move to new locations because instructions may be added, deleted or changed compared to the original code. Hence the targets of CTIs must be changed to their new locations. Doing so is easy for direct CTIs, since their targets are available in the CTI itself; the target can be changed to its new address in the output binary. However for indirect CTIs, the target may be computed many instructions before at an *address creation point* (ACP). It is impossible to find all possible ACPs for each CTI using dataflow analysis since they may be in different functions and/or propagated through memory (memory is not tracked by dataflow analysis.) Hence existing rewriters require relocation information to identify all possible ACPs. All ACPs must be present in relocation information since ACPs are precisely the list of addresses that need relocation during linking.

SecondWrite has devised technologies to rewrite binaries without relocation information. Details are in [26]; here we briefly summarize the intuition of our method. Rather than trying to discover ACPs, our basic method relies on inserting run-time checks at indirect CTIs that translate the old target to its corresponding new address using metadata tables that store such translations for all possible old branch and call targets. Aggressive alias analysis on the indirect CTI target is used to prune the list of such possible targets to a small number. Further, compile-time optimizations are applied when possible to reduce the number of run-time checks. The result is a method than can rewrite arbitrary binaries without relocation or symbolic information with very low overhead. The rewriter can then perform security enforcement on arbitrary binaries for the first time.

**Achieving 100% speculative code coverage** A key challenge in binary rewriters is discovering which parts of the code section in the input binary are definitely code, and thus should be rewritten. This is complicated since code sections often contain embedded data such as literal tables and jump tables which if rewritten by mistake will result in an incorrect program. The only way to be sure a portion of the code section is indeed code is to find a control-flow path from the entry point of execution to that portion. However portions of code may be reachable only through indirect control-transfer instructions (CTIs). Unfortunately the precise value set of CTI targets cannot be discovered statically in all cases; hence not all code may be discovered. Existing rewriters may not discover all the code, yielding incomplete code coverage – undiscovered code cannot be rewritten, and thus security cannot be enforced on it.

SecondWrite overcomes this problem by speculatively rewriting portions of the code segment which cannot be determined to be surely code, thus achieving 100% speculative code coverage. The detailed scheme is in [26]; but the intuition is that portions of the code segment which cannot be proven to be code are speculatively disassembled as if they are code anyway. If the speculative code turns out to indeed be code at run-time, then it is executed, achieving 100% speculative code coverage. Instead, if the speculative code arose from disassembling data bytes, that incorrect speculative code will never be executed since control will never transfer to it at run-time; preserving correctness. Instead the data is accessed from a copy of the original binary maintained in the rewritten binary. Maintaining this code copy increases code-size, but not the I-cache footprint since only the data portions of it are actually accessed, thus run-time is not affected. Since machines today have vastly more resources than even a few years ago, an increase in code size without increasing run-time is tolerable, especially given the payoff of being able to rewrite any binary.

**Rewriting to high-level intermediate representation (IR)** Unlike SecondWrite which represents programs in the high-level compiler IR, existing rewriters represent the binary using binary-like low-level code in the rewriter, making the program harder to analyze and modify. For example, high-level program features required for some security schemes, such as function arguments and return values, are not apparent in the binary. Further, existing rewriters retain register and memory accesses as-is, unlike SecondWrite which replaces both by symbolic accesses. Having memory accesses is problematic since it forces the layout of memory to be retained exactly in the rewritten binary, preventing modifications and optimizations of the stack and global segments, and additions to the stack segment. This too is inconvenient for security check insertion since such checks may allocate their own stack memory in some cases.

SecondWrite overcomes these programs by representing the binary code internally in compiler IR. Our method, described in [4], relies primarily on two technologies. First, high-level program features such as functions, and their arguments and return values are discovered from the binary using deep static analysis. Second, registers and memory locations are replaced by symbols as in high-level programs, allowing easy compiler modification of the memory allocation. With the resulting high-level IR, security checks become easy to apply.

## 5 Methods

One of the contributions of this paper is that we have carefully chosen a set of complementary and effective schemes that, taken together, achieve the goal of defending against all types of buffer overflow attacks at the lowest combined run-time cost. The totality of our schemes protect against not only the commonly known stack buffer overflow into return addresses, but is much more general than that, in that they protect against buffers on the global, stack and heap segments from overflowing onto a variety of code pointer locations that are possible in any data segment, including return addresses, function pointers, indirect branch pointers, *longjmp* buffers, and base pointers[5].

We implement our scheme by adding various passes that operate on high-level IR inside our binary rewriter. Our overall scheme consists of a number of components that we describe in detail in this section.

*Stack Canary Insertion* The first component of our scheme is the simplest. LLVM provides the ability to insert stack canaries during code generation. Utilizing this capability allows us to provide nearly the same level of protection to an un-protected binary as StackGuard [8] would provide when given an application's source code.

Essentially, a random canary value is generated at run-time and placed on the stack during a function's prologue. In the function epilogue, the value stored on the stack is compared with the random canary value for this process. If there is any difference, execution is halted as the canary value has been corrupted.

---

[5] Base pointers are not code pointers but lead to a similar vulnerability [31].

*Base Pointer Elimination* The *old base pointer* which resides on the stack is a data pointer that points to the base of the parent function's stack frame. Compilers sometimes introduce it since it makes it convenient to restore the stack pointer at the end of the function and to address different stack locations with the same offset even as the stack grows and shrinks in the function. When it is present in the input binary, it introduces a vulnerability just as dangerous as a code pointer [31]. This is because the old base pointer can be attacked by building a fake stack frame with a return address pointing to attack code, followed by overflowing the buffer to overwrite the old base pointer with the address of this fake stack frame. Upon return, control will be passed to the fake stack frame which immediately returns again redirecting flow of control to the attack code.

Given our unique use of LLVM IR in SecondWrite, the elimination of the base pointer in the output binary becomes a simple matter even when the input binary has base pointers. LLVM is an optimizing compiler and the binaries produced by LLVM are highly optimized. One common optimization applied by modern compilers on the x86 platform is to free up the EBP register for register allocation by removing the base (or frame) pointer. We used this LLVM pass to eliminate the base pointer from the binary.

When the base pointer is eliminated by LLVM, any attack relying on overwriting the base pointer is immediately prevented. There will be no base pointer for an attacker to modify. While corruption of the stack may still occur if an attacker overflows a buffer in order to attempt to overwrite the base pointer, no attack will be successful.

*Return Address Protection* Given that stack canaries as inserted by LLVM do not provide the same level of protection as the ProPolice mechanism that comes with GCC, we decided to implement a more complete solution similar to the protection scheme in StackShield [30], that protects against corruption of the return address. The basic idea of our return address protection scheme is as follows:

1. During the function prologue, push the return address of the current function in a return address stack implemented in a global data structure. For multi-threaded applications, multiple "shadow" stacks are maintained.
2. In function epilogue, compare the current return address on the stack with the value popped from the top of the return address stack.
3. If there is any difference between these values, execution is halted.

This simple scheme will detect if the return address has been modified either directly or indirectly. We implemented this scheme as it is relatively simple and protects against both direct and in-direct modifications of the return address. It also requires no modification of the stack layout and prevents modifications of the return address through buffer overflows in the heap or global segments.

Two challenges with this scheme are as follows. First, its overhead might be significant since every function has an associated security overhead incurred every time it is called. We found this overhead to be especially significant for recursive functions since they tend to short-running. Second, the size of the return address stack might be significant for deeply nested recursive functions, and we would have to bound it a-priori, which is hard to do.

We applied an optimization for relieving this problem which we call the *return address check optimization*. We observed that this protection mechanism is only necessary if a function contains a write to a stack buffer since return addresses only exist on the stack. This is hard to determine without symbolic information, so we conservatively try to prove that a function only has directly addressed memory references to constant addresses. If it finds any indexed write (base + runtime-variant offset), then it conservatively assumes that it could be a buffer write, and disables the optimization. If all writes are provably non-indexed writes to a constant offset, it enables the optimization, *i.e.*, the protection mechanism is turned off in the function. Thus the optimization saves on run-time overhead without sacrificing any protection.

We found this optimization surprisingly effective since it works best for small leaf functions in the call graph, and for recursive functions, which happen to be precisely the functions dynamically called most frequently. During our experimental evaluation of our scheme, of the many recursive functions we found, every one of them had its check optimized away. This is unsurprising since recursive functions tend to be short running, and unlikely to allocate stack arrays (although they may access portions of global arrays, such as in quicksort, but those still are optimized.) As a result of the optimization, the run-time overhead for scheme is greatly reduced, and the required return address stack depth is also greatly reduced. Of course, the overflow of the return address stack is not an error as we add extensions to it on the heap upon overflow, which slows execution, but is extremely rarely invoked even for small return address stack sizes of (say) 256 addresses.

*Function Pointer Protection* One common attack method used by attackers is to overwrite a function pointer so that when it is de-referenced, code of the attacker's choosing will be executed. In a binary executable, function pointers will appear as indirect calls. Thus, another component of our scheme concentrates on protecting all indirect calls and branches similar to how function pointers are protected in StackShield [30].

Our scheme adds checking code before all indirect calls and branches. A global variable is declared at the beginning of the data segment and its address is used as a boundary value. The checks inserted before any indirect call or branch ensure that the target of the indirect call or branch points to memory below the address of the global boundary variable. If the target points above the address of this global boundary variable then execution is halted.

An assumption in the above scheme is that a process follows the standard UNIX layout with the data segment above the code segment. This scheme does not protect against return-to-libc attacks since the target of the indirect call will still be within the code segment.

*Protection for* longjmp *buffers* The paired functions *setjmp* and *longjmp*, present in most C and C++ libraries, provide a means to alter a program's control flow in addition to the usual subroutine call and return sequence. First, *setjmp* saves the environment of the calling function (say *foo()*)into a data structure, and then *longjmp* in another function (say *bar()*) can use this structure to jump back to the point it was created, at the *setjmp* call. As a result, execution will return from *bar()* to *foo()* even when *foo()* is not the immediate parent of *bar()*. A typical use for *setjmp/longjmp* is exception handling.

The data structure used by *setjmp* for saving the execution state is referred to as a *jmp_buf*. Within this structure, enough information is stored to restore a calling environment. In particular, one member of this structure saves the value of the program counter which is used when restoring the calling environment. An attack method used by attackers is to overwrite the value of the program counter stored in the *jmp_buf* structure after a call to *setjmp* and before a call to *longjmp*. If this happens, control will be transferred to an address of the attacker's choosing when the *longjmp* is executed. Our method for defending against attacks of this kind is as follows:

1. Create a hash table within the global segment of the rewritten binary. Protect the hash table with write-protected (via *mprotect()*) guard pages, to mitigate attacks against it.
2. After each call to *setjmp* store the current value of the program counter in the *jmp_buf* structure into the hash table.
3. Before a call to *longjmp* get the current value of the *jmp_buf* structure that will be used. Attempt to perform a lookup in the hash table for the value of the program counter.
4. If the lookup in the hash table fails, then the value of the program counter has been modified and so we abort; otherwise execution continues

We expect the run-time overhead of this scheme to be very low in practice, since *setjmp* and *longjmp* calls are very rare. To the best of our knowledge, this scheme is the first protection scheme designed to protect against longjmp buffer attacks in the manner described. We intend to extend our scheme to cover the *ucontext_t* buffers and the *getcontext(), setcontext(), swapcontext()* API that is meant eventually to replace the *setjmp/longjmp* API.

## 6 Experimental Evaluation

We now present and discuss experimental results from our evaluation of our system. First, we examine the effectiveness of our security schemes as implemented in SecondWrite on a set of security benchmarks previously proposed by Wilander and Kamkar [31] for evaluating the effectiveness of buffer overflow defenses. Second, we examine how effective our scheme is in protecting against real-world attacks on widely-used real code (not benchmarks). Third, we examine the overheads of both the binary rewriter and our security scheme on some SPEC2006 and other benchmarks.

**Synthetic Results** In order to test how effective our scheme is, we utilized the benchmarks provided by Wilander and Kamkar [31]. Twenty buffer overflow attack forms were developed, in order to evaluate the effectiveness of tools available at the time that aimed to mitigate buffer overflow attacks. The attack forms covered every combination of buffer overflow attacks on global, stack, and heap buffers overflowing to a return addresses, base pointers, function pointers, and *longjmp* buffers. An attack form is defined as a combination of a technique, location, and an attack target. Of the twenty attack forms, we obtained the source code to only eighteen of these (*i.e.,* the other two were not available to us for evaluation). We then compiled the programs into binary code which we then rewrote using SecondWrite. Our schemes in SecondWrite successfully defended against all attack forms in the Wilander and Kamkar benchmarks.

**Real World Attacks** Ultimately, the success of our scheme depends on whether or not attacks that are observed in the real world can be prevented or not. Two real-world attacks were tested.

The first application we tested was GHTTPD – an HTTP server. This web server has a stack buffer overflow vulnerability in its logging function [15]. We obtained an exploit for GHTTPD which overflows a stack-based buffer and corrupts the return address. Using the return address protection component of our scheme, we were able to protect the return address and prevent the attack that uses the buffer overflow vulnerability to corrupt the return address. When our scheme is enabled, the return address corruption is detected when the attack occurs and the application is aborted.

The second application we tested was another HTTP server named CoreHTTP. This application contains a buffer-overflow vulnerability where it fails to adequately check user-supplied data before copying it to an insufficiently sized buffer [14]. We obtained an exploit for this application and applied our protection scheme to the application. Again, when our protection scheme is enabled, the attack is detected and the application is aborted.

**Binary Rewriting Overhead** A subset of SPEC benchmarks and other benchmarks were selected to substantiate the performance of our binary rewriter. The benchmarks were selected at random, and are limited only by the criteria that they are correctly rewritten by our still-early prototype. Table 1 lists the set of benchmarks that are used in the experiments. All the benchmarks are compiled with gcc 4.4.1. At this point, Secondwrite is not mature enough to rewrite large real-world commercial applications which are hence not included; debugging is ongoing. There are no fundamental limitations we know of in rewriting such programs.

| Application | Source | Lines of C Source Code |
|---|---|---|
| lbm | SpecFP2006 | 1155 |
| art | OMP2001 | 1914 |
| mcf | SpecInt2006 | 2685 |
| libquantum | SpecInt2006 | 4357 |
| sjeng | SpecInt2006 | 13847 |
| hmmer | SpecInt2006 | 35992 |
| h264 | SpecInt2006 | 51578 |

**Table 1.** Application Characteristics

In the first experiment, all binaries executed correctly after rewriting thus demonstrating SecondWrite's robustness. The standard suite of LLVM optimization passes ran without any changes in SecondWrite. These include CFG simplification, global optimization, global dead-code elimination, inter-procedural constant propagation, instruction combining, condition propagation, tail-call elimination, induction variable simplification and selective loop unrolling.

Besides correctness, the next most important metrics are the run-time speedup or overhead of the rewritten binaries versus the input binaries. For this paper, we study the performance of our rewriter on already optimized binaries. Figure 2 shows the normalized execution time of each rewritten binary compared to an input binary produced using GCC with the highest available level of optimization (-O3 flag). The results are mixed, with most benchmarks nearly breaking even or showing a small slowdown, one benchmark showing a larger slowdown of 20%, and one benchmark actually shows a speedup of 16%. The average is 2.7% slowdown.[6]

We consider this near break-even performance on highly optimized binaries a good result for three reasons:

- Our initial goal was not necessarily to get a speedup, but to generate correct IR without without introducing too much overhead. This would enable the IR to be a starting point for various custom compiler transformations we wanted to perform thereafter, such as automatic parallelization or security as covered in this paper. Ultimately, these optimizations determine the utility of the rewriter.
- These numbers represent our first-cut implementation devoid of any attempt at producing a better IR more geared towards optimization. We believe these numbers can be substantially improved with more detailed IR and are exploring several related avenues.
- We have currently not implemented any custom serial optimizations that might improve performance further, such as the inter-procedural versions of common sub-expression elimination and loop-invariant code motion, changing the compiler-enforced calling convention for registers for better run-time, and more aggressive inlining. We believe

---

[6] Rewriting unoptimized input binaries produced using GCC -O0 yields an average speedup of 27% using SecondWrite (not shown) due to its optimizations.

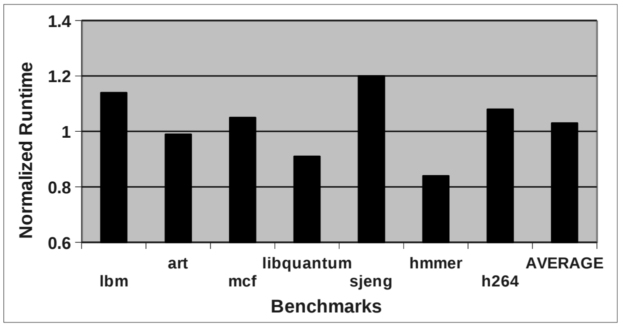

**Fig. 2.** Normalized runtime of rewritten binary as compared to optimized input binary (runtime=1.0).

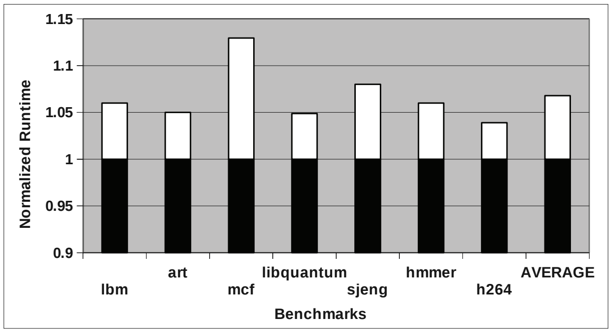

**Fig. 3.** Normalized runtime of rewritten binary with security scheme added.

these optimizations hold promise as the inter-procedural optimization abilities of current compilers are very limited compared to their intra-procedural performance.

One additional advantage of the binary rewriter is that it accumulates optimizations across two compilers—rewritten binaries have an optimization if it is either present in the compiler that produced the input binary, or in the rewriter. In our case, if either GCC or LLVM had an optimization, the output binary should have it. This is why, for example, one of our rewritten binaries (*hmmer*) had a 16% speedup versus the input binary. Although GCC with the -O3 optimization flag is known to produce good code, in some cases it missed promoting structure fields to registers whereas LLVM did, explaining the speedup in *hmmer*. With better IR and more aggressive optimizations, we expect to see more consistent speedups in output binaries in the future.

**Security Related Overheads** The overhead of the security schemes was measured on the same applications as used for measuring the overhead of the binary rewriter. The results are presented in Figure 3 and show overhead versus rewritten binaries without security schemes inserted. As seen, the average run-time overhead of 6.7% introduced by the protection scheme is low.

## 7 Conclusions

We have presented a new mechanism using an advanced binary rewriter that allows end users to retrofit powerful security features into third-party, binary-only software. The particular mechanisms we used are well known, and some have been partially implemented in other tools. Our system will allow end-users to retrofit program-level security protections for the first time in a highly customizable manner according to their needs and environment.

We demonstrated the effectiveness of our mechanism via experimental evaluation, begining with the benchmarks developed by Wilander and Kamkar. We successfully mitigated all the attack forms in the benchmarks. We then went on to demonstrate how our mechanism successfully defends against multiple real-world attacks. We also measured the overheads of our binary rewriter in isolation and then we showed what the overhead of adding the security mechanism to a binary is. In both cases, we demonstrated that the overhead introduced is quite low.

Future work involves extending the binary rewriter to work on more substantial applications and demonstrating that the mechanism defends against more real-world attacks and to better handle multi-threaded code and the new *ucontext_t* API. Other interesting avenues for future research are software diversification and self-healing techniques using the binary rewriter we have developed.

**Acknowledgements** This work was supported by the Air Force, DARPA and the NSF through Contracts AFRL-FA8650-10-C7024, AFOSR-MURI-FA9550-07-1-0527, DARPA-FA8750-10-2-0253 and NSF-CNS-09-14845, respectively. Any opinions, findings, conclusions or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, the Air Force, DARPA, or the NSF.

## References

1. Smashing the stack for fun and profit. Phrack magazine 7(49) (1996)

2. http://communities.vmware.com/docs/DOC 2601: List of VMWare White Papers
3. Abadi, M., Budiu, M., Erlingsson, U., Jigatti, J.: Control-flow integrity. In: Proceedings of the $12^{th}$ ACM Conference on Computer and Communications Security (CCS). pp. 340–353. ACM (2005)
4. Anand, K., Smithson, M., Kotha, A., Elwazeer, K., Barua, R.: Decompilation to Compiler High IR in a Binary Rewriter. Tech. rep., University of Maryland (November 2010), `http://www.ece.umd.edu/~barua/high-IR-technical-report10.pdf`
5. Boyd, S., Kc, G., Locasto, M., Keromytis, A., Prevelakis, V.: On The General Applicability of Instruction-Set Randomization. IEEE Transactions on Dependable and Secure Computing (TDSC) 7(3) (July–September 2010)
6. Bruening, D.: Efficient, transparent, and comprehensive runtime code manipulation. Ph.D. thesis (2004)
7. Cowan, C., Beattie, S., Johansen, J., Wagle, P.: PointGuardTM: Protecting pointers from buffer overflow vulnerabilities. In: Proceedings of the $12^{th}$ Usenix Security Symposium (2003)
8. Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., Hinton, H.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In: Proceedings of the $7^{th}$ USENIX Security Symposium. pp. 63–78. USENIX Association (1998)
9. Cowan, C., Wagle, P., Pu, C., Beattie, S., Walpole, J.: Buffer overflows: Attacks and defenses for the vulnerability of the decade. In: Proceedings of DARPA DISCEX. p. 1119. Published by the IEEE Computer Society (2000)
10. Eto, H., Yoda, K.: propolice: Improved Stack-smashing Attack Detection. Transactions of Information Processing Society of Japan 43(12), 4034–4041 (2002)
11. Eustace, A., Srivastava, A.: Atom: a flexible interface for building high performance program analysis tools. In: Proceedings of the USENIX Technical Conference. pp. 25–25 (1995)
12. Foster, J.: Buffer Overflow Attacks: Detect, Exploit, Prevent. Syngress Media Inc. (2005)
13. Hollingsworth, J.K., Miller, B.P., Cargille, J.: Dynamic program instrumentation for scalable performance tools. In: Proceedings of the Scalable High-Performance Computing Conference. pp. 841–850 (1994)
14. http://www.securityfocus.com/bid/25120/info: CoreHTTP Http.C Buffer Overflow Vulnerability
15. http://www.securityfocus.com/bid/5960/info: ghttpd log() Function Buffer Overflow Vulnerability
16. Hu, W., Hiser, J., Williams, D., Filipi, A., Davidson, J., Evans, D., Knight, J., Nguyen-Tuong, A., Rowanhill, J.: Secure and practical defense against code-injection attacks using software dynamic translation. In: Proceedings of the USENIX Conference on Virtual Execution Environments (VEE) (2006)
17. Kiriansky, V., Bruening, D., Amarasinghe, S.: Secure execution via program shepherding. In: Proceedings of the $7^{th}$ USENIX Security Symposium (2002)
18. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization (GCO). pp. 75–87 (2004)
19. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI). pp. 190–200 (2005)
20. Nanda, S., Li, W., Lam, L.C., Chiueh, T.: BIRD: Binary Interpretation using Runtime Disassembly. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO). pp. 358–370 (2006)
21. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM SIGPLAN Notices 42(6) (2007)
22. Rescorla, E.: Security Holes...Who Cares? In: Proceedings of the $12^{th}$ USENIX Security Symposium. pp. 75–90 (August 2003)
23. Romer, T., Voelker, G., Lee, D., Wolman, A., Wong, W., Levy, H., Bershad, B., Chen, B.: Instrumentation and optimization of Win32/Intel executables using Etch. In: Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop (1997)
24. Schwarz, B., Debray, S., Andrews, G., Legendre, M.: Plto: A link-time optimizer for the Intel IA-32 architecture. In: Proceedings of the Workshop on Binary Translation (WBT) (2001)
25. Shacham, H., Page, M., Pfaff, B., Goh, E., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: Proceedings of the $11^{th}$ ACM conference on Computer and Communications Security (CCS). pp. 298–307 (2004)
26. Smithson, M., Anand, K., Kotha, A., Elwazeer, K., Giles, N., Barua, R.: Binary Rewriting without Relocation Information. Tech. rep., University of Maryland (November 2010), `http://www.ece.umd.edu/~barua/without-relocation-technical-report10.pdf`
27. Solar Designer: ”return-to-libc” attack. Bugtraq Mailing List (August 1997)
28. Srivastava, A., Edwards, A., Vo, H.: Vulcan: Binary transformation in a distributed environment. Tech. Rep. MSR-TR-2001-50, Microsoft Research (2001)
29. Van Put, L., Chanet, D., De Bus, B., De Sutter, B., De Bosschere, K.: Diablo: a reliable, retargetable and extensible link-time rewriting framework. In: Proceedings of the IEEE International Symposium On Signal Processing And Information Technology. pp. 7–12 (December 2005)
30. Vendicator: Stack shield technical info file v0.7. (2001), `http://www.angelfire.com/sk/stackshield/`
31. Wilander, J., Kamkar, M.: A comparison of publicly available tools for dynamic buffer overflow prevention. In: Proceedings of the $10^{th}$ Network and Distributed System Security Symposium. pp. 149–162 (2003)
32. Witten, B., Landwehr, C., Caloyannides, M.: Does open source improve system security? IEEE Software 18(5), 57–61 (2001)