Rapid Vulnerability Mitigation with Security Workarounds

Zhen Huang Gang Tan

Department of Computer Science and Engineering
Pennsylvania State University
zhen.huang@psu.edu, gtan@cse.psu.edu

***Abstract*—The existence of pre-patch windows allows adversaries to exploit vulnerabilities before they are patched. Prior work has proposed to harden programs with security workarounds to enable users to mitigate vulnerabilities before a patch is available. However, it requires access to the source code of the programs. This paper introduces RVM, an approach to automatically hardening binary code with security workarounds. RVM statically analyzes binary code of programs to identify error-handling code in the programs, in order to synthesize security workarounds. We designed and implemented a prototype of RVM for Windows and Linux binaries. We evaluate the coverage and performance of RVM on binaries of popular Windows and Linux applications containing real-world vulnerabilities.**

I. Introduction

Software users are plagued by emerging vulnerabilities on a daily basis. According to cvedetails [17], a popular database for disclosed vulnerabilities, 15,830 CVE entries for vulnerabilities have been created this year, which is a 7% increase over last year. Patching vulnerabilities is effective and necessary, but there exists *pre-patch windows* which allow adversaries to exploit vulnerabilities before patches are created and applied [19]. Because pre-patch windows are mainly caused by the inevitable time and effort required to manually analyze vulnerabilities, then create and test patches for the vulnerabilities, it is unrealistic to expect pre-patch windows can be reduced considerably or eliminated unless automated patch generation is widely adopted.

To protect applications in the absence of patches, many tools have been proposed to harden applications in order to raise the bar for adversaries to exploit vulnerabilities. The level of security guarantees that they provide is usually inversely proportional to the extent of the information required for target applications and the cost in terms of the loss of performance or functionality. The vast majority of these tools aim to protect applications without any functionality loss [18], [24], [26]. Unfortunately this objective usually causes high performance overhead and thus it is difficult for users to adopt them in practice.

Security Workaround for Rapid Response (SWRR) is a recently proposed approach to hardening applications with negligible or no performance overhead in exchange of little or minor functionality loss [19]. SWRRs follow the same principle behind the commonly used configuration workaround [1], [3], [6], [7] that sacrifices certain functionality to mitigate vulnerabilities rapidly before patches are available. The principle is that users are willing to exchange minor functionality loss for security.

While SWRRs is a promising solution to allow users to exchange minor functionality loss for the rapid protection for severe vulnerabilities, it is designed and generated in the form of source code and instrumented into the source code of a target application [19]. As a result, it can be used only on open-sourced applications or by software vendors who have access to source code.

In this paper, we propose an approach called RVM that applies SWRRs directly to binary code in order to address the limitation. RVM automatically generates SWRRs in the form of binary code and incorporates them to a target application without access to the source code of the application. This allows regular users to use RVM to mitigate vulnerabilities swiftly in close-sourced applications.

RVM faces two major challenges in applying an SWRR to binary code: 1) identifying error-handling code in binaries, and 2) generating and instrumenting binaries to incorporate an SWRR.

First, one important reason that configuration workaround has wide adoption in practice is that it achieves *unobtrusiveness*, the property that it affects only the functionality relevant to vulnerable functions. To achieve similar unobtrusiveness offered by configuration workaround, Talos relies on the success of finding existing error-handling code of a target application to generate SWRRs [19]. However, it is challenging to follow the same approach to finding existing error-handling code in binary code. We discuss this challenge in details in Section III-A and Section III-B.

Second, generating SWRRs in the form of binary code requires more considerations, such as calling conventions, than in the form of source code, and instrumenting binary code to incorporate SWRRs requires knowing the location in binary executable files at which SWRR is instrumented and considering whether the instrumentation requires relocating other existing code and data. The challenge is discussed in more details in Section III-C.

RVM addresses the first challenge by using a novel approach in finding error-handling code and by adopting static program analysis specifically designed for binary code. Particularly it uses API error specifications automatically mined from online API documentations as the basis, then by following error propagation to find error-handling code in binary code. To facilitate analyzing binary code, it conducts analysis on VEX IR code lifted from binary instructions.

It addresses the second challenge by generating SWRRs using code cloning and finding the location of instrumentation using an approach oblivious to file formats.

In summary, we make the following contributions in this paper:

- We propose an approach to finding error-handling code in existing applications by mining API error specifications from API documentations and leveraging error propagation.
- We designed and implemented a prototype of RVM that can apply SWRRs in the form binary code on x86-64 Windows and Linux applications. The code for the prototype is available at https://gitlab.com/zhenhuang/rvm.git.
- We evaluated the coverage of SWRRs produced by our prototype through a case study on using SWRRs to mitigate real-world vulnerabilities, and also evaluated the performance of our prototype.

II. Background and Related Work

A. SWRR

Security Workaround for Rapid Response (SWRR) is a mechanism proposed to mitigate software vulnerabilities rapidly [19]. It is a simple code snippet instrumented into a target program to prevent a vulnerability from being triggered. Because it disallows the execution of the vulnerable code at the granularity of functions, it can prevent any inputs including polymorphic inputs designed to trigger the vulnerability, and thus stops any form of further attacks such as ROP [28], albeit at the cost of losing the functionality provided by the instrumented function.

We illustrate how an SWRR mitigates a vulnerability using an example vulnerability. Listing 1 presents the vulnerable code adopted from a real-world vulnerability CVE-2011-4362 in `lighttpd`, a popular web server. The vulnerable function `base64_decode` decodes an input base64 string using a lookup table `base64_reverse_table` with the input string as the index. Owing to the lack of a proper check on whether the input string can be used as valid index, an adversary can craft malicious input strings to cause out-of-bounds table lookup and thus abnormal termination of `lighttpd`.

The design of SWRR highlights two key features: simplicity and unobtrusiveness. First, as shown in Listing 2, an SWRR is merely a simple return statement instrumented to the beginning of function `base64_decode` so that no vulnerable code will be executed and thereby no inputs can trigger out-of-bounds lookup of table `base64_reverse_table`. The SWRR effectively neutralizes the vulnerability by disabling `lighttpd`'s base64 decoding.

Second, by returning a NULL, the SWRR achieves unobtrusiveness by indicating an error to the caller function `http_auth_basic_check`, so that `lighttpd` can properly handle this error. Essentially this leads to the rejection of

```
unsigned char* base64_decode(char *out,char *in) {
    unsigned char *result = out;
    int ch, i = 0, j = 0;
    ...
    for (...) {
        ch = in[i];
        ch = base64_reverse_table[ch];
        ...
        result[j] = ch;
    }
    if (ch == base64_pad && i % 4 == 0)
        return NULL;
    return result;
}

// returns 0 on failure; 1 on success
int http_auth_basic_check(...) {
    ...
    if (!base64_decode(...)) {
        log_error_write("decodeing base64-string failed");
        return 0;
    }
    ...
    return 1;
}
```

Listing 1. Example vulnerability, adopted from CVE-2011-4362 in `lighttpd`.

```
unsigned char* base64_decode(..., char *in) {
    /* SWRR inserted at top of function */
    return NULL;

    /* original function body */
    ...
}
```

Listing 2. An SWRR instrumented into the vulnerable `base64_decode` function in Listing 1.

any HTTP basic authentication that requires base64 decoding. However, other functionality is intact so `lighttpd` can continue to process other forms of authentications.

Because SWRRs are simple and require only the information about which return values should be used to indicate an error, they can be mechanically synthesized and instrumented into a target program and save the time and effort of human developers. As a result, they can dramatically reduce the pre-patch window [19] and used as a rapid response to mitigate software vulnerabilities.

B. Hardening Binary Code

Many approaches [25], [26], [32], [34], [35] have been proposed to hardening binary code of programs to protect the programs from malicious attacks such as control flow hijacking, malicious web browser extensions, Return-Oriented Programming (ROP) [28] and Counterfeit Object-oriented Programming (COOP) [27].

Some of them implement various forms of Software-based Fault Isolation (SFI) [30], [33] or Control Flow Integrity (CFI) [14] on binary code. Among them, Lockdown enforces CFI on ELF dynamic shared objects and rewrites binaries using Dynamic Binary Instrumentation (DBI) [26]. NaCl adopts SFI and provides an execution sandbox for native binary code, executed as part of web browser extensions [35].

CFI and SFI can provide comprehensive protection against different types of exploits. However, they incur from 5 to 20% performance overhead. In contrast, SWRR instrumented by RVM incurs neglect or no performance overhead.

Another form of binary code hardening is code randomization. To hinder ROP attacks, SmashGadgets [25] randomizes binary code using techniques such as atomic instruction substitution, instruction reorder, and register reassignment; as a result, program instructions intended to be used as gadgets [28] for these attacks can no longer be used. While code randomization is effective in thwarting ROP attacks, it does not prevent other types of exploits such as regular stack buffer overflow and control flow hijacking [15].

Compared with existing binary hardening tools, RVM instruments SWRRs that are designed to be simple and effective in preventing vulnerabilities from being triggered by disabling the execution of entire vulnerable functions. RVM gives users a choice between security protection and minor functionality loss in response to severe vulnerabilities.

C. *Inferring Error-Handling Code*

Inferring error-handling code in programs has mainly two purposes. First, knowing the locations of error-handling code allows tools designed to find bugs in error-handling code to focus on these locations [20], [22], [22], [31]. Second, error-handling code indicates error return values that can be used to synthesize SWRRs for security [19].

To infer error-handling code, most of these tools rely on information about the error return values of API functions called by a target program. They either make some unsound assumptions on error return values, such as that all non-zero return values are error return values as long as zero is also one of the return value [22], or depend on other tools or documentations to provide such information [20], [31]. Particularly LFI works on binary code. However, it does not perform interprocedural error propagation like RVM does.

In contrast, APEx uses characteristics include the number of statements, function calls, and paths to differentiate error-handling code from other code [21]. Furthermore, Talos adopts a two phase approach [19]. In the first phase, it relies on heuristics such as “error-handling code often calls error logging functions before returns a constant” to generate an initial list of error return values. In the second phase, it follows error propagation in the target program to identify other error return values.

III. Problem Description and Challenges

As discussed in Section II-A, SWRRs is a simple, unobtrusive, and effective mechanism to rapidly mitigate software vulnerabilities. Talos has demonstrated how to automatically synthesize and instrument SWRRs for C/C++ programs [19]. However, it is challenging to apply SWRRs directly to binary code.

In order to achieve unobtrusiveness, i.e. not affecting functionality irrelevant to the vulnerable function, an SWRR needs to return an error value to invoke existing error handling code to reject the input that triggers the corresponding vulnerability, and to recover from the error in order to be able to process the next input. For example, the SWRR for vulnerability CVE-2011-4362 in `lighttpd` needs to return a NULL from the vulnerable function `base64_decode` so that its caller would be able to propagate the error and eventually `lighttpd` would reject the request that triggers the execution of the vulnerable function.

To identify error return values that can be used by SWRRs, Talos finds existing error handling code of a target program by using five heuristics in static program analysis. It first uses two main heuristics, *error-logging function* and *NULL return*, to find a initial set of functions that contain error-handling code, then uses two extension heuristics to identify error propagation in the program and thus finds error-handling code in other functions based on the initial set of functions. While Talos achieves success on its coverage on open-sourced programs, we find using these heuristics on close-sourced program pose new challenges.

A. *Main Heuristics*

The main heuristics play a critical role for the success of Talos, because they not only achieve a significant coverage by themselves but also form the basis for other heuristics. However, we find that the main heuristics have two major limitations when applying to close-sourced programs.

First, many close-sourced programs, especially those designed only for Windows, make little use of error-logging functions. One possible reason is that Windows API functions follow a well-established standard set of error return values [11] and thus examining a return value is sufficient to reveal what kind of error occurs, without the need of logging the error. As a result, the error-logging function heuristic does not find much error-handling code for close-sourced programs on Windows. Second, returning a NULL from a function that normally returns a valid pointer usually indicates an error occurred. So an SWRR might return NULL as an error return value for a function that returns a pointer. However, it is non-trivial to decide whether a function returns a pointer without the type information of the program. We will describe how we address these limitations in Section V-B and Section V-C.

B. *Extension Heuristics*

The extension heuristics rely on data flow analysis to identify error propagation in a target program. For example, the error raised in function `base64_decode` is propagated to its caller function `http_auth_basic_check` via the return of NULL at line 12, and further propagated to the caller function of `http_auth_basic_check` via the return of 0 at line 21, as shown in Listing 1.

To recognize such error propagation, a data flow analysis needs to follow exactly how the return value from `base64_decode` is checked in its caller at line 19 and what constant values are returned by its caller when the check succeeds and when the check fails, respectively.

However, data flow analysis on binary code in general is more challenging than on source code due to compiler optimization, mixed use of registers and variables, and lack of type information. We will describe our solution to address this challenge in Section V-D, Section V-E, and Section V-F.

```
int http_auth_basic_check(...) {
  400684:        push    %rbp
  400685:        mov     %rsp,%rbp
  400688:        sub     $0x20,%rsp
  ......         ......
  4006b6:        callq   400672 <base64_decode>
  4006bb:        test    %rax,%rax
  4006be:        jne     4006dd
  4006c0:        mov     $0x400798,%edi
  4006c5:        callq   400646 <log_error_write>
  ......         ......
  4006d6:        mov     $0x0,%eax
  4006db:        jmp     4006e2
  4006dd:        mov     $0x1,%eax
  4006e2:        leaveq
  4006e3:        retq
```

Listing 3. Assembly code compiled from function `http_auth_basic_check` in Listing 1.

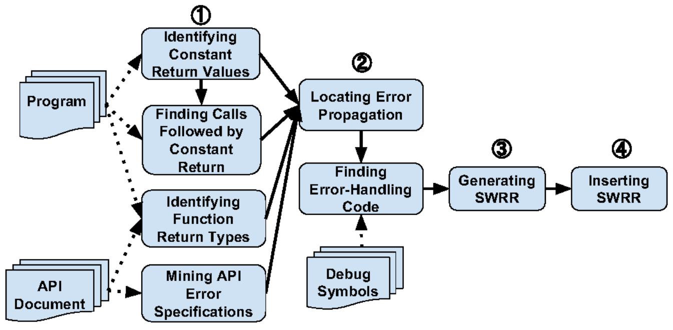
①
Identifying Constant Return Values
②
Program
Locating Error Propagation
Finding Calls Followed by Constant Return
③
④
Finding Error-Handling Code
Generating SWRR
Inserting SWRR
Identifying Function Return Types
API Document
Mining API Error Specifications
Debug Symbols

Fig. 1. Workflow of RVM: each rounded rectangle represents a step in RVM; each circled number denotes a phase that consists of one or more steps underneath the circled number; dotted arrows denote input data, while solid arrows denote the order of steps.

C. Generating and Instrumenting SWRRs

An SWRR in the form of source code is simply a C/C++ `return` statement. But such a statement is implemented as different instructions for different architectures. Even for the same architecture, different calling conventions can result in different instructions. For example, `stdcall` convention dictates that caller functions allocate stack space for the call and callee functions clean up the allocation, while `_cdecl` convention dictates the opposite. An SWRR for these two different calling conventions must behave differently to follow the calling conventions, and thus RVM must generate different instructions accordingly. Therefore generating SWRRs in the form of binary code needs to consider the target architecture and the calling convention.

Instrumenting an SWRR in the form of source code is simply inserting the source code of the SWRR at the beginning of the source code of a target function; so only the starting source code line of the function is required. However, instrumenting an SWRR in binary code requires knowing the location of the function in binary code, which depends on the format of the binary code. We describe how our solution generates SWRRs in the form of binay code and instruments SWRRs in binary code in Section V-H and Section V-I, respectively.

IV. RVM

We design an approach called RVM (Rapid Vulnerability Mitigation) to address the challenges described in Section III. In this section, we describe a typical usage scenario of RVM.

RVM is designed to be used by end-users of a target program to rapidly mitigate a known but not yet patched vulnerability in the target program. To illustrate its usage, we use a real-world vulnerability CVE-2011-4362 in function `base64_decode` of `lighttpd` web server, which is presented in Listing 1 .

When a user of `lighttpd` knows that vulnerability CVE-2011-4362 in function `base64_decode` exists in her version of `httpd`, from the security notice of the vendor of `lighttpd` [3], she may have several choices.

First, she can download the official source code patch, apply it to the source code of `lighttpd`, build the source code, and install the newly built binary. This requires that she understands how to apply the source code patch and build `lighttpd` from the source code.

Second, she can apply the configuration workaround disclosed in the security notice [3] by disabling the `mod_auth` module in `lighttpd`, in order to mitigate the vulnerability. Unfortunately her `lighttpd` will no longer be able to use any form of HTTP authentication once the `mod_auth` module is disabled.

Third, she can wait for her OS vendor to issue a patched `lighttpd` as an OS update, if she is unable to follow the first choice and the second choice. But this can take a long time. As a matter of fact, the vendor of RedHat Linux applied the patch to the `lighttpd` shipped with the OS six months after the source code patch was released [4].

Last, she can run RVM with the vulnerable function name, `base64_decode`, and the location of the binary code of `lighttpd`, to automatically generate an SWRR for this function and to apply the SWRR to the binary code of her `lighttpd`. After being applied, this SWRR mitigates the vulnerability and only disables the basic HTTP authentication, which depends on base64 decoding; other forms of HTTP authentications can still be used.

Compared with other choices, RVM is a solution that offers the benefits of rapid response, easy-to-use, and unobtrusiveness together, which are desirable in most situations.

V. DESIGN AND IMPLEMENTATION

A. Overview

RVM generates an SWRR in binary code for a target program and inserts the SWRR into the target program in four phases, as shown in Figure 1.

The first phase takes the binary code of a program as input, and is made up of four steps:

- *identifying function return types* – outputs the return type for functions in the program, whose details are described in Section V-C
- *identifying constant return values* – outputs constant function return values, whose details are described in in Section V-D

- *finding calls followed by constant return* – outputs a list of function call that are followed by a return of a constant value, referred to as *constant returns*, whose details are described in Section V-E
- *mining API error specifications* – mines API documentation to find API error specifications, i.e. error return values for API functions, whose details are described in Section V-B

The second phase finds error return values for functions in the target program. It uses several heuristics to find error-handling code in a target program and takes the following information as input.

- Constant values returned by every function in the program
- Function calls that are followed by a return of a constant value, and each of these function calls and its corresponding return must be guarded by the same condition
- API documentation that describes error return values for API functions
- Debug symbols that include the name and entry address for every function in the program

Most of the above information is prepared by the first phase. The second phase outputs the constant values used as error return values for functions in the program. It consists of two steps: *locating error propagation* and *finding error-handling code*, which will be described in details in Section V-F and Section V-G, respectively.

The third phase generates an SWRR. It takes as input the name of the vulnerable function and the error return values for functions in the program, which is the output of the second phase, synthesizes an SWRR for the function and outputs the SWRR along with the entry address of the function. The name of the vulnerable function can usually be found in the CVE report or bug report of a vulnerability [19]. This phase contains the step of *generating SWRRs*. We describe it in more details in Setion V-H.

The fourth phase takes the entry address of the vulnerable function and the SWRR generated in the third phase as input, and outputs a modified binary code of the program with the SWRR instrumented. This phase includes one step: *inserting SWRRs*. We describe it in details in Setion V-I.

We base RVM on angr [29], a static analysis framework for binary code. and Talos [12], [19], a tool generating and instrumenting SWRRs for C/C++ programs. Talos is implemented as a standalone frontend that analyzes LLVM IR code generated from C/C++ source code, and a backend that generates SWRRs in the form of C/C++ source code, and instruments SWRRs into C/C++ programs. We implement the steps in phase 1 that analyzes binary code using angr as a new frontend for Talos, and extend the backend of Talos to implement the steps in all other phases to generate SWRRs in the form of binary code and instrument SWRRs into binary code.

Becaue RVM works on VEX IR code [23] lifted by angr from binary code, RVM can be used for binary code of all the architectures supported by angr, including 32-bit and 64-bit versions of ARM, MIPS, PPC, and x86. However, we use only the 64-bit x86 (x86-64) architecture and assembly code in this paper for ease of description.

B. Mining API Error Specifications

As we discussed in Section III, programs running on Windows do not use error-logging functions as much as open-sourced programs. As a result, we cannot rely on calls to error-logging functions to identify an initial set of functions that have error-handling code for binary code.

To find an alternative approach to identify such initial set of functions, we conducted an informal analysis of Windows programs and libraries to study their error-handling code. We find that they make intense use of Windows API functions and they usually check whether an error occurred by examining the return value of calls to these API functions, if these functions have return values. By studying the official documentation for Windows API functions, we find that the vast majority of them can return an error value. And they follow a standard set of error return values, called *system error codes* [11]. We also find that similarly Linux programs and libraries rely heavily on OS system calls and API functions implemented by common libraries such as libc, both of which have official documentations.

Note that previous work has also considered using documented error return values of API functions, often called an *error specification*, to find error-handling code. Some existing work [20], [31] relies entirely on an error specification provided as input. However, as far as we are aware, we are the first to mine API specifications from API documentation, and then uses error propagation to identify error return values defined in a program, particularly in binary code.

Because Windows API documentations are commonly posted online, we developed a web crawler to crawl Windows API documentation websites and mine API error specifications. The crawler is built on scrapy [5]. For our prototype of RVM, our crawler supports Microsoft Windows API documentation [10].

In contrast, Linux systems usually deploy API documentations as man pages on users' computers. These man pages are usually stored in compressed format on the file system and only temporarily uncompressed when an user views them. To mine Linux API documentation, we developed a simple text analysis tool that decompress each man page and searches for documentation on API functions.

We describe our results on mining Windows and Linux API specifications in Section VII-A.

Because API documentations are usually written in respect to source code, using it on binary code requires to match functions in the binary code with those described in the documentation. We note that the goal of RVM is to harden user applications, and the fact that it is a common practice to ship debug symbols for not only user applications but also even an entire OS such as Windows and different flavors of Linux [8], [9], [13]. Some existing binary hardening tools such as Lockdown [26] and REINS [34] also rely on debug symbols. We also note that we only require the debug symbols to get the mapping from function names to entry addresses, and we could switch to a different approach to get this mapping without relying on debug symbols.

C. Identifying Function Return Types

To locate error propagation in a target program, RVM requires information on which functions have a pointer return value, e.g. `char *`. For programs with source code, function return types can be found in function prototype declarations. However, stripped binary code does not have such information.

RVM takes advantage of the information on API function prototypes mined from API documentation. It follows call chains to identify function return types, starting with return types of API functions and propagating these return types to callers of these API functions in the program.

The propagation maintains a list of function return types, which is initially filled with API function return types, and performs an iteration on every function of the program whose return type is unknown. In each iteration, it finds a function that has a pointer return type if its return value is derived from the return value of a call to another function that has a pointer return type.

Whenever the return type of a new function has been found, the propagation adds the function return type to the list and starts a new iteration. The propagation terminates when no new function return types can be found in an iteration.

D. Finding Constant Return Values

This step finds constant return values for functions in a target program. These constant return values are then used to find existing error-handling code in the target program. It takes the CFG and the VEX IR code of the program as input, both of which are generated by angr.

To find constant return values, we need to locate where in the code a return value is assigned, which we refer to as an *assignment site*, and where in the code a return value is passed back to the caller of a function, which we refer to as a *return site*. We also need to distinguish the case when a constant value is assigned as a return value and the case when a non-constant value is assigned.

It is common for a function to have more than one return value. The binary code of a function is often organized in a way to save the number of return instructions. For example, function `http_auth_basic_check` can return 0 or 1, as shown in Listing 3. It has one return site, a single `ret` instruction at line 16, and two assignment sites, two `mov` instructions with register `eax` as destination at line 12 and line 14, respectively.

To find out which constant values are used as return values, RVM links each return site with its corresponding assignment sites using a backward intraprocedural static analysis. Each return site can be trivially identified by looking for `ret` instructions. Each assignment site is defined as the reaching definition of register `rax`, i.e. the last assignment to register `rax` preceding a return site in the control flow.

RVM identifies constant-value operands used in assignment sites as return values. Because a function may assign a constant value to a variable and then use the variable as its return value, RVM uses the reaching-definition analysis to find if a return value stored in a variable is indeed a constant.

For each function in the target program, this step outputs a list of *constant return sites*, each of which is denoted as a pair of a return site and its associated assignment site. Each pair is denoted as a tuple of (`return_address`, `assignment_address`, `return_value`).

We illustrate how the step works by using the example function `http_auth_basic_check` in Listing 3. This step first identifies that the function has one return site at line 16. It then checks if there is any assignment site in the same basic block containing line 16 and it finds that line 14 is an assignment site, because the `mov` instruction at line 14 assigns a constant value `1` to register `eax`. After that, it iterates through all the predecessor of the basic block containing the return site in the control flow graph, and checks if there is any assignment site in each predecessor. When it checks the predecessor starting at line 10, it finds that line 12 is an assignment site that assigns constant `0` to register `eax`.

At last, it outputs a *list of constant return sites* that includes (`0x4006e3`, `0x4006d6`, `0`) and (`0x4006e3`, `0x4006dd`, `1`) for the example function `http_auth_basic_check`.

E. Finding Calls Followed by Constant Returns

This step finds function calls that are immediately followed by returns of constant values. It takes the list of pairs of return site and assignment site generated from the last step as input, and outputs a *list of function calls followed by constant returns*.

We consider that a function call is followed by an assignment site if two conditions are satisfied: 1) the basic block containing the assignment site post-dominates the basic block containing the function call and 2) the two basic blocks have the same control dependency. We define two basic blocks having the same control dependency if they are control dependent on the same condition check and they are on the same branch following the check. And we exclude control dependency introduced by loop conditions from consideration.

Line 20 and line 21 in Listing 1 are an example of a function followed by a constant return, because the two lines are control dependent on the condition check on line 19 and they are on the `if` branch following the condition check.

For a given function, RVM first finds the function's assignment sites from the list of pairs of return sites and assignment sites. It then marks each of the function's assignment site with the control dependency of the assignment site. After that, it iterates through all the function calls in this function and checks if the function call is control-dependent and, if so, whether the function call is post-dominated by any one of the assignment sites that has the same control dependency. If it finds a function call followed by a constant return, it adds a tuple of (`function_call_address`, `assignment_address`, `control_dependency`) to its output list.

For example, this step would check the function call at line 6 and at line 10 for function `http_auth_basic_check` in Listing 3. Because the function call at line 6 is not control dependent on any condition checks, it excludes the function call from further consideration. For the function call at line 10, it finds that 1) the function call is control dependent on the condition check at line 7, 2) the assignment site at line 12 post-dominates the function call, 3) the assignment site is also control dependent on line 7, and 4) the function call and the assignment site are on the same branch following line 7.

As a result, this step outputs a list for function `http_auth_basic_check` that contains only one function followed by a constant return denoted as a tuple (`0x4006c5`, `0x4006d6`, `0x4006bb`).

F. Locating Error Propagation

Following error propagation to find error return values is critical for the coverage of SWRRs [19]. As a basis for the step of finding error-handling code, this step takes the CFG and the VEX IR code for the target program as input, and outputs information on how error is propagated.

It differentiates two ways of propagating error return values: *direct propagation* and *translated propagation*. For the former, it outputs the list of function calls whose return value is directly propagated. For the latter, it outputs not only the list of function calls whose return value is translated before being propagated but also the way of the translations.

Direct propagation. A function can make a function call and simply use the return value of the function call as its own return value. In this case, the function making the function call would have the same error return values as the callee function of the function call. One example is a return statement such as `return(foo())`, in which the return value of the call to `foo` is directly used as the return value of the caller function.

This step identifies direct propagation by looking for a function call and a following return, between which there is no modification of the function return value. The way to modify the function return value is dependent on the ABI used by the program. For example, register `rax` is commonly used as the function return value for x86-64; so in this architecture a modification of the return value is defined as a call to a function that has its own return value, i.e. not a `void` function, or a direct modification of register `rax`.

Similar to the two conditions used to find calls followed by constant returns, described in Section V-E, this step determines that a direct propagation must satisfy three conditions: 1) the basic block containing a return site post-dominates the basic block containing a function call, 2) the two basic blocks have the same control dependency, and 3) there is no modification to the function return value on the path from the function call to the return site. Note that a `void` function might also satisfy such conditions. But this will not affect finding error-handling code, because any return value propagation would stop at a `void` function.

Translated propagation. A function can "translate" the return value of a function call into a different value and use it as its own return value. A translation consists of two actions: 1) a conditional check on the return value of a function call, and 2) a return statement that returns a constant value on one of the branches guarded by the conditional check. As described in Section V-D, we refer to the latter as a constant return site.

One example of such translation occurs in function `http_auth_basic_check` in Listing 1, which translates the return value NULL from the call to `base64_decode` into 0. The example translation consists the conditional check on line 19 and a return of constant 0 on line 21. Because the step of identifying constant return values already takes care of outputting constant return sites including line 21, this step only needs to output the conditional check of the return value of the function call at line 19. Particularly its output includes the condition used in the conditional check.

It defines a conditional check on the return value of a function as two actions: a function call and a following conditional check on the return value of the function call. Identifying such conditional checks poses the following challenges particularly for binary code:

1) the return value can be propagated to a series of local variables, on the last of which the check is performed;
2) there are various ways to store a local variable, such as in a register or on the stack;
3) the check of the return value can be performed against a constant value or a local variable that contains a constant value;
4) there are various ways to implement the same check in binary instructions, e.g. checking if a return value is zero can be implemented in several ways such as `test rax, rax` and `or rax, rax` on x86-64;
5) there are various ways to assign a constant value in binary instructions, e.g. setting register `rax` on x86-64 to zero can be implemented in several ways such as `xor rax, rax` and `mov 0, rax`

Building our analysis on VEX IR code lifted from binary code significantly helps us address these challenges. Particularly different binary instructions that are commonly used to perform the same operations such as value comparison are translated into the same VEX IR instruction. For example, the x86-64 instructions `test rax, rax` and `cmp rax, 0` are translated into the same VEX IR instruction `CmpEQ64`. This makes it easier for us to address the last two challenges.

The first three challenges could be addressed with copy propagation and reaching definition analysis. Unfortunately no prior work on binary analysis can provide a variable recovery and reaching definition analysis on binary code with the same quality as those on source code. To be efficient and conservative, our prototype of RVM performs copy propagation and uses path-insensitive data flow analysis to locate a definition or assignment for a register or a variable, and will terminate the analysis if there are more than one definition to the same register or variable and these definitions occur on different paths. Essentially this can cause an under-approximation for identifying translated propagation.

G. Finding Error-Handling Code

This step takes the output from phase 1 as input, and outputs a list of error return values for each function in the target program. Particularly, its input includes the API documentation on error return values for API functions, the debug symbols, the call graph of the program, the information on error propagation, and the following information for each function of the program:

- list of constant return values
- list of function calls followed by constant returns
- a CFG

From the input, it extracts the error return values for API functions from the API documentation, and uses heuristics to identify error-handling code in each function of the program. The two key heuristics, error-logging functions and NULL returns, are discussed in Section III.

It then follows the NULL return heuristic to find functions whose return type is pointer. To ensure that NULL can be safely returned from these functions, it verifies if any caller of the function checks the return value against NULL. If so, it considers NULL as an error return value for these functions.

After that, it follows the error-propagation information to find error return values for other functions.

H. Generating SWRRs

With a given name of a vulnerable function, RVM aims to generate one or more SWRRs to protect this function. There are three different cases: 1) when RVM finds an error return value for the function, RVM generates one SWRR for the function; 2) when RVM cannot find an error return value for the function but finds error return values for all the callers of the function, RVM generates one SWRR for each of the caller function; 3) when RVM cannot find an error return value for neither the function nor all of its callers, RVM cannot generate an SWRR to protect the function.

As described in Section II-A, an SWRR consists of a `return` statement. So RVM needs to generate binary instructions corresponding to the `return` statement. For example, a `return` statement is implemented as a `ret` instruction and the return value is passed back in register `eax` or `rax` for x86 and x86-64 platforms, respectively. Depending on the calling convention used by the program, the `ret` instruction may also take a constant operand that specifies the number of bytes on the stack that should be cleaned up.

Consequently, RVM needs information on the calling convention and the error return value of the function to be protected to generate an SWRR for a function.

Because different architectures can use different application binary interface (ABI) for function calls, RVM needs to generate an SWRR specifically for each architecture. For example, x86 and x86-64 use the `ret` instruction, while ARM uses the `bx lr` instruction. Our current prototype focuses on x86 and x86-64 platforms so it generates an SWRR as a `mov eax` or `mov rax` instruction with the error return value as its operand and a following `ret` instruction with an optional operand used for cleaning up the stack.

To find out whether the `ret` instruction needs an operand and what constant value should be used as the operand, one approach is to examine the existing `ret` instruction in the function. Instead, RVM chooses a simpler approach by cloning the existing `ret` instruction in the function, based on the information provided by `angr` on the address and length of the `ret` instruction.

I. Inserting SWRRs

A common approach to insert new instructions into binary code safely is to use a binary instrumentation tool such as DynInst [16], because the insertion can involve complex operations such as relocating existing instructions and/or data and finding the binary file offset corresponds to the entry address of the function, which requires taking into account different formats of binary code, such as PE and ELF.

But we note that inserting an SWRR does not require preserving the original instructions of the target function, because they will not be executed anyway. As a result, an SWRR can be inserted by overwriting the starting instructions of the function with the instructions of the SWRR without the need for relocation. This will work unless the size of the function is smaller than the size of the SWRR instructions. Because an SWRR consists of only two instructions that occupy either six or seven bytes, it is rare to have a function too small to hold an SWRR.

And finding the binary file offset corresponding to function entry address can be achieved by using a brute-force approach that searches the instructions of the function in the entire binary file, without the need of knowing the format of the binary code. Although this approach can be inefficient, it is applicable to most if not all binary code formats.

VI. Limitations

RVM relies on API error specifications automatically mined from online API documentation and local man pages to identify error return values that can be used to generate SWRRs. While the online documentation and local man pages are usually up-to-date, they might be inconsistent with the version of the API functions installed on a particular computer. And sometimes they might not get updated as fast as new versions of API functions are released. Although this kind of inconsistency can cause issues for developers who work with these API functions, we note that RVM only requires error return values of these API functions, which are rarely changed in practice.

Our prototype of RVM assumes that the target binary does not use self-modifying code and is unpacked, so that it could use a straightforward brute-force approach to locate the instructions of a function in the binary file without taking into account different formats of binary files. In our future work, we plan to use a more reliable approach that follows the format of binary files.

VII. Evaluation

In this section, we first present our results of mining API error specifications from online documentations, and then report the coverage of SWRRs produced by RVM. We focus on the coverage of SWRRs produced by RVM, because the security guarantee of SWRRs is not affected by whether the SWRRs are produced in the form of binary code or source code. After that, we illustrate how SWRRs instrumented by

API Interface	# Sources	# Category	# Functions	# Header Files
Windows	22,973	707	15,359	5,071
Linux	5,142	N/A	3,455	385

TABLE I. API ERROR SPECIFICATIONS MINED BY RVM. FOR WINDOWS, THE COLUMN "SOURCES" REFERS TO URLS. FOR LINUX, IT REFERS TO MAN PAGES.

CVE#	OS	App.	Binary	Size	# Func.
CVE-2006-3730	Windows	IE	webvw.dll	133KB	585
CVE-2006-4071	Windows	OS	gdi32.dll	281KB	1,499
CVE-2011-4362	Linux	lighttpd	mod_auth.so	76KB	75

TABLE II. BINARIES THAT HAVE REAL-WORLD VULNERABILITIES.

RVM mitigate real-world vulnerabilities using case study. Finally we present the performance of RVM on analyzing binaries, generating SWRRs, and instrumenting the SWRRs into binaries.

For all our evaluations, we use a workstation that has an Intel Core i7-7700 CPU running at 3.60GHz and 16GB RAM. The workstation runs Ubuntu 16.04 desktop operating system on a 2TB 7200 RPM SATA hard drive.

A. API Error Specifications

As described in Section V, we build a web crawler to crawl online Windows API documentations and a text analyzer to mine local Linux man pages to mine API error specifications. In this section, we present our results on mining API error specifications.

Note that we mine error specifications directly from either online API documentations or local man pages, rather than header files. However, these API documentations are indeed generated by software vendors from header files, as described in the documentations. So we count the number of header files from which these API documentation are generated by using the information in the documentations. This gives us a rough idea that how many header files need to be mined to retrieve the same information if we mine the header files for API functions.

As shown in Table I, our web crawler visited 22,973 URLs and identified the error specification for 15,359 Windows API functions, which belong to 707 different categories according to the documentation. By contrast, our text analyzer searched through 5,142 man pages and found the error specification for 3,455 Linux API functions. Mining from these URLs and man pages can be considered as equivalent from 5,071 and 385 header files, respectively.

B. Coverage

We use real-world vulnerabilities in popular Windows and Linux applications for our evaluation. For each vulnerability, we choose to use the particular binary that contains the vulnerable function to apply SWRRs. The vulnerabilities are listed in Table II, which also shows the type of operating system (OS), the name of the application and the binary, the size of the binary and the number of functions that the binary has.

The results on the coverage of SWRRs produced by RVM for these binaries are shown in Table III. The column

Binary	Protected	API Spec.	Pointer.	Prop.	Indirect
webvw.dll	55.0%	0.7%	0.3%	36.2%	17.8%
gdi32.dll	75.5%	16.4%	0.0%	30.8%	28.3%
mod_auth.so	77.3%	9.3%	0.0%	0.0%	68.0%
AVERAGE	69.3%	8.8%	0.1%	22.3%	38.0%

TABLE III. COVERAGE OF SWRRS PRODUCED BY RVM.

"Protected" shows the percentage of the functions that can be protected by SWRRs. The column "API." and "Pointer." show the percentage of the functions that whose error-handling code are identified using API error specifications and pointer return types, respectively. The column "Prop." presents the percentage of functions whose error return value is identified via following the error propagation in the binary. Lastly the column "Indirect" presents the percentage of functions that are protected indirectly by SWRRs in all of their caller functions.

We can see that on average RVM can apply SWRRs to 69.3% of the functions in these binaries. Using API error specifications and pointer return types allows RVM to identify 8.8% and 0.1% of the functions respectively. While following error propagation helps identifying the error return values for 22.3% of the functions, 38.0% of the functions have to be protected by SWRRs in their caller functions.

C. Case Study

We use an Internet Explorer vulnerability CVE-2006-3730 [2], shown in Table II, as a case study to illustrate how RVM can be used to rapidly provide protection for users of the unpatched Internet Explorer.

This is an integer overflow vulnerability in the `setSlice` method of an ActiveX object contained in the `webvw.dll` shared library used by Internet Explorer. By crafting a malicious web page that contains a call to this vulnerable method with a specific argument, an adversary can trigger the vulnerability and execute arbitrary code with the permissions of the user when the user browses the web page with Internet Explorer. Because exploits for this vulnerability had been released before a patch was available, users and system administrators were advised to apply a configuration workaround that disables the use of this vulnerable ActiveX control completely.

When RVM is used to apply SWRRs to mitigate the vulnerability, it first finds that `setSlice` calls a Windows API function `DSA_SetItem`, which returns TRUE on success and FALSE on failure, and `setSlice` uses the return value from the API function as its own return value when the API function returns FALSE. As a result, RVM determines that FALSE or 0 is also an error return value for `setSlice`.

Because this function uses the `stdcall` calling convention, it must free up the stack space allocated by its caller when it returns to the caller. However, RVM does not need to concern about the calling convention in generating the SWRR for this function, because it uses instruction cloning to copy the `ret` instruction of the function as that of the SWRR. It then synthesizes a `mov 0, eax` instruction that assigns 0 (FALSE) as the function's return value, and appends the cloned `ret` instruction of the function as an SWRR for this function, as shown below.

Binary	angr	Phase 1	Phase 2	Phase 3 & 4	Total
webvw.dll	15s	394s	0.3s	20s	429.3s
gdi32.dll	16s	3107s	0.6s	26s	3149.6s
mod_auth.so	0.7s	13s	0.1s	1s	14.8s
AVERAGE	10.6s	1171.3s	0.3s	15.7s	1197.9s

TABLE IV. PERFORMANCE OF RVM: ALL EXECUTION TIME IS MEASURED IN SECONDS.

```
mov 0, eax
ret 0x38
```

After this, RVM locates the start of the function in the binary file by searching for the first 32 bytes of the instructions of the function in the binary file. Once it locates the offset of the instructions, i.e. the start of the function, it overwrites the start of the function with the instructions of the SWRR after making a backup of the original binary file.

D. Performance

We measure the execution time that RVM takes to analyze a binary, generate an SWRR, and instrument the SWRR into the binary. The results are presented in Table IV.

We separate the execution time of the underlying `angr` frame work and RVM to find out how much execution time does RVM add on top of the execution time of `angr`. The column "angr" contains the execution time for `angr` to generate the underlying data structures used by RVM, including the loading of a binary code and the construction of a CFG. The column "Phase 1" and "Phase 2" shows the execution time of phase 1 and phase 2 of RVM, respectively. The column "Phase 3 & 4" includes the execution of both phase 3 and phase 4. Finally the column "Total" presents the total execution time for all phases of RVM. And all the execution times are reported in seconds.

As we can see, phase 1 takes the vast majority of the total execution time, as it performs intense program analysis on a binary to identify information required to find error-handling code. In total it can take RVM nearly an hour to apply an SWRR to mitigate a vulnerability. However, the user does not need to interfere with the execution of RVM after starting it, because all the phases are completely automated. And our prototype has not been optimized, and we believe its performance can be considerable improved after optimization.

VIII. CONCLUSION

This paper presents RVM, an approach that rapidly mitigates un-patched vulnerabilities in binary code using static program analysis and binary rewriting. RVM applies Security Workarounds for Rapid Response (SWRR) to gracefully disable the execution of vulnerable functions in order to prevent the vulnerabilities from being exploited. From our evaluation on binaries that contain real-world vulnerabilities, we find that RVM can apply SWRRs to 69.3% of the functions of a binary on average. This is comparable to the coverage achieved by prior work that applies SWRRs on source code.

REFERENCES

[1] "Apache httpd Vulnerability Workaround," http://mail-archives.apache.org/mod_mbox/httpd-users/201408.mbox/%3CCAC=HunseOneq3nOoVbSSADMPVgTpDeihHYOu+95b66fLUT2Qow@mail.gmail.com%3E.

[2] "Microsoft Windows WebViewFolderIcon ActiveX Control setSlice() Integer Overflow Vulnerability," https://tools.cisco.com/security/center/viewAlert.x?alertId=11787.

[3] "out-of-bounds read due to signedness error," https://download.lighttpd.net/lighttpd/security/lighttpd_sa_2011_01.txt.

[4] "Red Hat Bugzilla Bug 758624," https://bugzilla.redhat.com/show_bug.cgi?id=758624.

[5] "Scrapy — A Fast and Powerful Scraping and Web Crawling Framework," http://scrapy.org.

[6] "Squid Range Headers Vulnerability Workaround," http://www.squid-cache.org/Advisories/SQUID-2014_2.txt.

[7] "Microsoft Security Advisory 3009008 - Vulnerability in SSL 3.0 Could Allow Information Disclosure," https://docs.microsoft.com/en-us/security-updates/securityadvisories/2015/3009008, 2015.

[8] "Debug Symbol Packages," https://wiki.ubuntu.com/Debug%20Symbol%20Packages, 2018.

[9] "INSTALLING DEBUGINFO PACKAGES," https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/developer_guide/intro.debuginfo, 2018.

[10] "Programming reference for Windows API," https://docs.microsoft.com/en-us/windows/desktop/api/index, 2018.

[11] "System Error Codes," https://docs.microsoft.com/en-us/windows/desktop/debug/system-error-codes, 2018.

[12] "Talos: a software tool that automatically generates and instruments SWRRs into target applications using static program analysis. ," https://github.com/huang-zhen/Talos, 2018.

[13] "Windows Symbol Packages," https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-symbols, 2018.

[14] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, 2005, pp. 340–353.

[15] Aleph One, "Smashing the stack for fun and profit," *Phrack Magazine*, vol. 7, no. 49, 1996.

[16] B. Buck and J. K. Hollingsworth, "An API for Runtime Code Patching," *The International Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 317–329, 2000. [Online]. Available: https://doi.org/10.1177/109434200001400404

[17] http://www.cvedetails.com, 2018, accessed: May, 2018.

[18] W. Huang, Z. Huang, D. Miyani, and D. Lie, "LMP: Light-weighted Memory Protection with Hardware Assistance," in *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ser. ACSAC '16. New York, NY, USA: ACM, 2016, pp. 460–470. [Online]. Available: http://doi.acm.org/10.1145/2991079.2991089

[19] Z. Huang, M. D'Angelo, D. Miyani, and D. Lie, "Talos: Neutralizing vulnerabilities with security workarounds for rapid response," in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 618–635.

[20] S. Jana, Y. J. Kang, S. Roth, and B. Ray, "Automatically detecting error handling bugs using error specifications," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pp. 345–362. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/jana

[21] Y. Kang, B. Ray, and S. Jana, "Apex: Automated inference of error specifications for c apis," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 472–482. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970354

[22] P. D. Marinescu and G. Candea, "Lfi: A practical and general library-level fault injector," in *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, June 2009, pp. 379–388.

[23] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *The 2007 ACM Conf. on Program-*

ming Language Design and Implementation (PLDI), Jun. 2007, pp. 89–100.

[24] B. Niu and G. Tan, “Modular control-flow integrity,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’14. New York, NY, USA: ACM, 2014, pp. 577–587. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594295

[25] V. Pappas, M. Polychronakis, and A. D. Keromytis, “Smashing the gadgets: Hindering return-oriented programming using in-place code randomization,” in *2012 IEEE Symposium on Security and Privacy*, May 2012, pp. 601–615.

[26] M. Payer, A. Barresi, and T. R. Gross, “Fine-grained control-flow integrity through binary hardening,” in *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9148*, ser. DIMVA 2015. Berlin, Heidelberg: Springer-Verlag, 2015, pp. 144–164. [Online]. Available: https://doi.org/10.1007/978-3-319-20550-2_8

[27] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. Sadeghi, and T. Holz, “Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications,” in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 745–762.

[28] H. Shacham, “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86),” in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, Oct. 2007, pp. 552–61.

[29] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, “SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis,” in *IEEE Symposium on Security and Privacy*, 2016.

[30] G. Tan, “Principles and implementation techniques of software-based fault isolation,” *Foundations and Trends in Privacy and Security*, vol. 1, no. 3, pp. 137–198, 2017.

[31] Y. Tian and B. Ray, “Automatically diagnosing and repairing error handling bugs in c,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 752–762. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106300

[32] V. v. d. Veen, E. Gktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, “A tough call: Mitigating advanced code-reuse attacks at the binary level,” in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 934–953.

[33] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, “Efficient software-based fault isolation,” in *ACM SIGOPS Operating Systems Review*, vol. 27, 1994, pp. 203–216.

[34] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, “Securing untrusted code via compiler-agnostic binary rewriting,” in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC ’12. New York, NY, USA: ACM, 2012, pp. 299–308. [Online]. Available: http://doi.acm.org/10.1145/2420950.2420995

[35] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, “Native client: A sandbox for portable, untrusted x86 native code,” in *2009 30th IEEE Symposium on Security and Privacy*, May 2009, pp. 79–93.