Pinpointing Vulnerabilities

Yue Chen
Florida State University
ychen@cs.fsu.edu

Mustakimur Khandaker
Florida State University
khandake@cs.fsu.edu

Zhi Wang
Florida State University
zwang@cs.fsu.edu

Abstract

Memory-based vulnerabilities are a major source of attack vectors. They allow attackers to gain unauthorized access to computers and their data. Previous research has made significant progress in detecting attacks. However, developers still need to locate and fix these vulnerabilities, a mostly manual and time-consuming process. They face a number of challenges. Particularly, the manifestation of an attack does not always coincide with the exploited vulnerabilities, and many attacks are hard to reproduce in the lab environment, leaving developers with limited information to locate them.

In this paper, we propose Ravel, an architectural approach to pinpoint vulnerabilities from attacks. Ravel consists of an online attack detector and an offline vulnerability locator linked by a record & replay mechanism. Specifically, Ravel records the execution of a production system and simultaneously monitors it for attacks. If an attack is detected, the execution is replayed to reveal the targeted vulnerabilities by analyzing the program's memory access patterns under attack. We have built a prototype of Ravel based on the open-source FreeBSD operating system. The evaluation results in security and performance demonstrate that Ravel can effectively pinpoint various types of memory vulnerabilities and has low performance overhead.

1. INTRODUCTION

Memory is the field of eternal arms races between attacks and defenses [59]. Commodity operating systems have employed exploit mitigation mechanisms such as data-execution prevention (DEP, aka $W \oplus X$) [19, 20], address space layout randomization (ASLR) [60], mandatory access control (MAC) [32], etc. Yet, attackers can always find a way with new vulnerabilities and new exploit techniques. Defenses like ASLR can often be bypassed by combining several exploits. For example, an attacker may exploit an information leak to de-randomize the victim process before launching a return-oriented programming (ROP [54]) attack to disable DEP, and then inject and execute the shellcode. In light of this, a timely response to new (zero-day) exploits is essential to defenses.

Many systems have been proposed to detect attacks. However, they often focus on detecting symptoms of attacks. A detected attack thus does not necessarily coincide with the targeted vulnerabilities. For example, system call (syscall) interposition tries to detect anomalies in the syscalls made by a protected program [27, 28, 30], based on the observation that an attacker eventually needs

ASIA CCS '17, April 02 - 06, 2017, Abu Dhabi, United Arab Emirates

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4944-4/17/04...$15.00

DOI: http://dx.doi.org/10.1145/3052973.3053033

```
1 int main(int argc, char *argv[])
2 {
3     char buf[16];
4     strcpy(buf, argv[1]);
5     printf("%s\n", buf);
6     return 0;
7 }
```

Figure 1: A simple program with a buffer overflow at line 4.

to make syscalls to perform "useful" malicious activities. Specifically, it models and monitors syscall sequences of that program. An intrusion alert is raised if an actual sequence deviates from the model. A detected anomaly in this case reveals neither the initial attack nor the vulnerabilities it targets. Similarly, the detection of control-flow hijacking may not concur with the vulnerabilities as well. For example, taint-based attack detection systems [12, 48] mark the untrusted inputs as tainted, and propagate those taints throughout the system. An attack is detected if the program counter (PC) becomes tainted. Control-flow integrity (CFI [1, 9]) can also detect such attacks: it instruments the program with inline reference monitors that check the program's run-time control flow against its pre-computed control-flow graph (CFG). A deviation from the CFG signals that the control-flow has been hijacked. However, neither taint- nor CFI-based systems can pinpoint the exploited vulnerabilities. This can be illustrated with the simple program in Figure 1 – they both detect the attack at line 6 where the control flow is first hijacked, but the actual vulnerability lies at line 4. Syscall interposition detects the attack even later, i.e., when an unexpected syscall is made. In brief, many attack detection systems fall short of revealing the targeted vulnerabilities.

A system that can not only detect attacks but also pinpoint the exploited vulnerabilities could greatly help us in the arms-race against attackers. First, it can significantly reduce the window of vulnerability. Developers often spend lots of time to reproduce and analyze reported attacks. This is usually a manual, time-consuming, and error-prone process as many attacks are hard to reproduce in the development environments. Second, it can automatically locate zero-day vulnerabilities, as long as the attacks can be detected. Many existing systems can detect zero-day attacks (i.e., they do not rely on the details of known attacks), including the previously mentioned syscall interposition and taint-/CFI-based systems. Lastly, locating vulnerabilities is an important first step towards automatic software repair and self-healing.

In this paper, we propose Ravel [1], a system that can pinpoint the targeted vulnerabilities from detected attacks. Ravel stands for "Root cause Analysis of Vulnerabilities from Exploitation Log." It consists of three components: an online attack detector, a record & replay (R&R) mechanism, and an offline vulnerability locator. R&R decouples the other two components so that the online attack detector can operate as efficiently as possible to minimize the performance overhead, and the offline vulnerability locator can employ

[1]**Ravel**: *to undo the intricacies of*: Disentangle ($\frac{\textit{Merriam}}{\textit{Webster}}$)

```
 1 int process_request(int conn_fd, struct Header *
     packet_header, char *buffer_to_send)
 2 {
 3     char buffer[MAX_BUF_LEN];
 4     size_t size;
 5     ssize_t recved;

 6     size = (size_t) min(packet_header->length,
           MAX_BUF_LEN);

 7     recved = recv(conn_fd, buffer, size);
 8     save_to_file(buffer, recved);
 9     send(conn_fd, buffer_to_send, size);

10     return 0;
11 }
```

Figure 2: A vulnerable function used as the running example. There is a buffer overflow in line 7 caused by the integer signedness error in line 6, and an information leak in line 9 caused by the same integer error.

multiple, time-consuming algorithms to improve its precision and coverage. As many attack detection techniques have been proposed, we leverage some existing light-weight detectors (program crashes and syscall interposition). Note that the development of attack detectors is orthogonal to the Ravel framework. New techniques can be easily employed by Ravel for better and faster attack detection. In this paper, we focus on the design of the overall framework of Ravel and the vulnerability locator, the main contributions of Ravel. The intuition behind the vulnerability locator is that exploiting a memory vulnerability often causes changes to the data flow, and the source or the destination of such a change provides a good approximation to the actual location of the vulnerability [11]. For example, `strcpy` in Figure 1 can overflow into the return address on the stack if `argv[1]` is longer than the `buf` size. This introduces a new data flow with `strcpy` as the source and the return statement (line 6) as the destination. In this case, the source actually points to the vulnerability we want to locate. However, data-flow changes in general can only provide a rough location of the vulnerability. Ravel further refines them with vulnerability-specific analysis to pinpoint common memory flaws such as integer errors, use-after-free, race conditions, etc. We have implemented a prototype of Ravel for the FreeBSD operating system (Release 10.2). Our experiments with standard benchmarks and various vulnerabilities in popular applications show that Ravel can pinpoint a variety of memory vulnerabilities, and it incurs only a minor performance overhead (about 2% for SPEC CPU 2006, NGINX, Apache, etc.). This demonstrates Ravel's effectiveness and practicality.

2. DESIGN OF RAVEL

2.1 System Overview

Given an attack, Ravel aims at automatically pinpointing the vulnerabilities it targets. There are many challenges in locating vulnerabilities (in addition to detecting attacks). We use the example code in Figure 2 to illustrate them. Many design decisions in Ravel are made to address these challenges. The code in Figure 2 is inspired by a real vulnerability in the popular NGINX web server [16]. Function `process_request` abstracts how the server processes a client's request. Specifically, the server reads the request from socket `conn_fd` (line 7) and logs it to a local file (line 8), it then sends back its response (`buffer_to_send`) to the client (line 9). The `packet_header` parameter points to the packet header previously received from the client. Its `length` field specifies how much data to receive from the client. Consequently, this field is under the attacker's control. To avoid overflowing the receive buffer, line 6 limits `length` by the buffer size. Unfortunately, this line has an integer signedness error. More specifically, `packet_header->length` has a type of `ssize_t` (i.e., signed `size_t`, an alias of `int`). If the attacker makes it negative, it will pass `min` without any change and be converted to a large positive number saved in `size`. This leads to a stack-based buffer overflow in line 7, a potential denial-of-service in line 8 (by filling the disk space), and an information leak in line 9 (by sending lots of data to the attacker). Note that `recv` normally returns any data currently available in the socket up to the requested size. The attack thus also controls how much data to receive and write in line 7 and 8, respectively. We use this code as a running example in the rest of this section. Even though the source code is used in these examples, Ravel works on program binaries and thus does not require the source code. However, if we do have the source code or the debugging information, we can easily map the located vulnerabilities in the binary to the source code.

This example demonstrates many challenges in pinpointing vulnerabilities. *First*, the manifestation of an attack does not necessarily reveal the real vulnerabilities. The real vulnerability in Figure 2, an integer signedness error, lies in line 6. Without this flaw, the rest of the function cannot be exploited. Therefore, line 6 is the root cause of those attacks. It is this line that the developer should fix. However, most attack detection systems fail to reveal this root cause because they look for anomalies in the program's behaviors and can only detect an attack when or after it has happened. For example, control-flow integrity [1] can detect a violation at line 10, and syscall interposition [30] only detects the attack when the payload is executing. Data-flow integrity [11, 57] can reach closer to the root cause but still cannot pinpoint it. In this paper, we define a data flow as a def-use relation between instructions [11]. Specifically, an instruction "defines" a memory location if it writes to that location, and an instruction "uses" a memory location if it reads from that location. Two instructions form a def-use relation if they write to and read from the same memory location, respectively. Anomalies in the data flow can help us identify two derived vulnerabilities in line 7 and 9 since they both introduce extra def-use relations. The root cause, nevertheless, is the integer error in line 6. To address this challenge, Ravel first uses a data-flow analysis to approximate the real vulnerability, and further refines the result by analyzing its details. *Second*, several vulnerabilities may co-exist together allowing multiple ways to exploit them. Figure 2 contains four vulnerabilities. The attacker may choose to take over the control flow by the buffer overflow, or dump the server's memory by the information leak (without triggering the buffer overflow since the attacker can control how much data to be received). Ravel needs to handle individual vulnerabilities as well as their combinations. *Third*, techniques to locate vulnerabilities often require analyzing the program's detailed memory access patterns, a prohibitively time-consuming technique without special hardware support. To be practical, Ravel has to address this important performance challenge. *Finally*, how to faithfully reproduce attacks is also challenging but may be indispensable for finding root causes.

Figure 3 shows the overall architecture of Ravel. Ravel consists of three components: an online attack detector, a record & replay (R&R) mechanism, and an offline vulnerability locator. The target process is executed under the control of a record agent (we call it the recorder for brevity). The recorder logs the complete execution history of the process for replaying later. Recent advances in R&R, such as eidetic systems [23], allow Ravel to continuously record the execution of a process with low performance and storage overhead. While recording, the attack detector monitors the execution

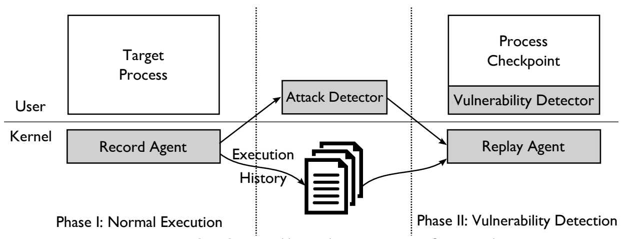

Figure 3: Overall architecture of Ravel

of that process for exploits and attacks. In order to capture real-world attacks, the recorder and the attack detector run on production systems. If an attack is detected, the execution log can be sent to the developer (likely on a different computer) for further analysis using the vulnerability locator. The vulnerability locator replays the recorded execution and performs a number of analyses to pinpoint the vulnerabilities. Specifically, it first uses a data-flow analysis to roughly locate the vulnerabilities and then performs vulnerability-specific analysis to refine the results. In this architecture, we use a combination of the general data-flow analysis and vulnerability-specific analyses to fulfill Ravel's precision requirements (the 1st and 2nd challenges), and leverage R&R to address the performance and reproducibility requirements (the 3rd and 4th challenges). In the rest of this section, we give details of these components.

2.2 Detecting Attacks

To locate vulnerabilities, Ravel first needs to detect and record the attacks. The attack detector thus plays an important role in Ravel. In order to handle real-world (zero-day) vulnerabilities, the attack detector has to run on the production system. This imposes strict performance and effectiveness requirements on the attack detector. However, Ravel is structured as an extensible framework. It can employ many attack detection techniques, such as syscall interposition and CFI. This is made possible by the design of Ravel. Specifically, the vulnerability locator deduces the rough location of an exploited vulnerability by searching for anomalies in the data flow and further refines that with detailed analyses. Therefore, it is sufficient for the vulnerability locator to know that the execution log contains certain (unknown) attacks. This minimal requirement allows Ravel to employ any attack detection technique as long as it is effective and has low performance overhead. We want to emphasize that the attack detector itself may provide very little help in locating vulnerabilities. For example, syscall interposition detects attacks only when the payload is executing, but the actual exploitation is hidden in the haystack of other executed instructions.

In our prototype, we employ two simple attack detection techniques – program crashes and syscall interposition [30]. They both have low performance overhead. With the wide-spread deployment of exploit mitigation techniques like $W \wedge X$ and ASLR, they are also more effective than before. For example, $W \wedge X$ prevents the injected malicious code from being executed. To address that, the attacker often uses return-oriented programming (ROP [54]) to change the process' memory permission with an unplanned syscall. This can be readily detected by syscall interposition. Moreover, ASLR often makes an exploit less stable, leading to more frequent crashes under attack. Both techniques can be easily integrated into Ravel. In particular, our implementation of syscall interposition validates both syscall sequences and parameters. Syscall interposition has been well researched [26, 30, 34, 39, 40, 43, 45, 46, 50], so we omit the details here. The derivative vulnerabilities in Figure 2 (line 7, 8, and 9) can be detected by checking syscall parameters and potentially by crashes if ASLR is supported. We plan to support more advanced detection techniques, such as CFI, in the future.

2.3 Record and Replay

Record & replay (R&R [23, 31, 41, 53]) plays an important role in Ravel: it bridges the performance gap between the attack detector and the (sluggish) vulnerability locator, making Ravel usable even in production systems, and it makes the attack reproducible for many times. How to faithfully reproduce in-the-wild bugs/attacks is a big challenge in the software development and maintenance. Ravel imposes two requirements on its R&R system. *First*, since the recorder runs on the production system, it should incur minimal performance and storage overhead. Additionally, it must be compatible with the attack detector without changing the program's normal execution. *Second*, the execution history is replayed by the vulnerability locator, which instruments the replayed execution with heavy-weight analyses. These analyses will make additional system calls (e.g., to allocate memory for the intermediate results) and consequently affect the memory layout of the replayed process. The replayer must isolate these side-effects to keep the replay faithful.

To fulfill these requirements, Ravel employs a process R&R system, which records and replays a single process or a group of related processes. The process R&R can be implemented either at the library level or the kernel level. Ravel chooses the latter because it is more secure in an adversarial environment — the library-level recording can be bypassed if the program/payload makes direct system calls, and the recorded history cannot be trusted if the process is compromised because the recorder exists in the (compromised) process' address space. Moreover, our R&R system assumes that the target program properly synchronizes its access to shared resources. Deterministically recording and replaying arbitrary programs that have race conditions could be very complicated and inefficient without hardware support [41]. If the program does have race conditions (i.e., bugs that should be fixed by developers), the replayed execution will deviate from the recorded history and can thus be detected. Lastly, Ravel records the complete execution history of the target process. To reduce the storage overhead, Ravel borrows various techniques from the Eidetic System [23], a practical always-on whole-system R&R system. For example, Ravel uses LZMA to compress the execution log and avoids logging the data that can be recovered from the environment, such as reading a static file.

2.3.1 Record

To faithfully replay a process, Ravel needs to record all the non-deterministic inputs to that process. These inputs include both the data and events from the external environment (e.g., network packets and signals) and the internal events (e.g., locks). Next, we describe in detail how these two types of inputs are handled by Ravel.

The syscall interface is an ideal location to intercept and record external inputs such as syscall returns, the user-space memory modified by a syscall, and signals. Syscall returns need to be recorded because they may affect the program execution. For example, a program may use its process id returned by `getpid` as one of the ingredients to generate random numbers. Syscall returns can also affect the control flow (e.g., error handling). It is rather straightforward to record syscall returns – we just need to log them in the execution history. A syscall may modify the user-space memory. For example, the `stat` syscall writes the file status into the user-provided memory. Most such syscalls explicitly define the structures of the exchanged data. If so, we simply save the modified memory after the syscall returns. In this way, we understand and retain the semantics of the modified data. However, syscalls like `ioctl` may write different amounts of data to the user space when given different parameters. Even worse, `ioctl` can be dynamically extended by a loaded kernel module. It is hard to record all the user-space memory written by these system calls. To address that, Ravel hooks the FreeBSD kernel's `copyout` function to record (and replay) the data copied to the user space during those syscalls. Similar to Linux's `copy_to_user` function, FreeBSD exclusively uses `copyout` to write data to the user-space. Signals can also introduce non-determinism to the recorded process. A signal can be either synchronous and asynchronous. A synchronous signal (e.g., SIGSEGV) is the result of exceptional program behaviors. There is no need to record this kind of signals because replaying the program will trigger the same exceptions. An asynchronous signal (e.g., an alarm) instead must be faithfully recorded and replayed. Since it is asynchronous, we can delay its delivery until a syscall return. This greatly simplifies the replay of asynchronous signals.

Some instructions can bypass the kernel and directly interact with the hardware. A typical example on the x86 architecture is the `RDTSC` instruction, which returns the CPU's current time-stamp counter. Some programs use the outputs of `RDTSC` for random number generation. Therefore, Ravel has to record the outputs of this instruction. However, `RDTSC` is by default an unprivileged instruction and can be executed by any user programs. To address that, we change the CPU's configuration (the `TSD` flag in the `CR4` register) to intercept the execution of `RDTSC` by user processes and record its outputs for the target process. The interception of `RDTSC` is turned off when the kernel switches to other processes. As such, there is no overhead for other processes.

The internal non-determinism comes mostly from accessing the shared memory. To avoid race conditions, the program should synchronize these accesses, say, by using locks. Without race conditions, it is sufficient for Ravel to record the order of processes (or threads) entering critical sections. Replaying the execution in the same order ensures that the shared memory is in the correct state for each critical section. To this end, we instrument the synchronization primitives in common libraries (e.g., the `pthread` library) to record and replay them in orders. Examples of these primitives include `pthread_mutex_lock`, `pthread_rwlock_wrlock`, `pthread_cond_broadcast`, `pthread_cond_signal`, `sem_wait`, `atomic_store`, `atomic_exchange`, etc. On the other hand, if the program does have race conditions (e.g., two threads modify the same data without synchronization), the replay will deviate from the recorded execution history. Ravel tries to detect race conditions when that happens.

2.3.2 Replay with Instrumentation

After Ravel detects an attack, it starts to replay the recorded execution to locate vulnerabilities. For most syscalls, such as `getpid` and `stat`, it is not necessary to re-execute them. Ravel just returns the recorded return values and updates the user memory if necessary. Similarly, network connections (sockets) are not recreated during the replay. Ravel directly returns the recorded data to the replayed process. Other syscalls, typically memory related one have to be re-executed. For example, most programs use `mmap` to allocate memory. The kernel may return a different block of memory during the replay. To address that, we pass the recorded memory address in the first parameter of `mmap`, which is a suggestion of the allocation address to the kernel. During our experiments, the kernel always accepts the suggestion and allocates the same memory.

During the replay, Ravel instruments the program in order to detect anomalies in the process' memory access patterns. We use dynamic binary translation (BT) for this purpose. As such, the same program binary can be used for both recording and replaying. The BT engine can interfere with the replay. For example, the engine needs to allocate memory for its own use (e.g., to cache the translated code). This may conflict with the memory layout of the recorded execution. The engine also makes extra syscalls, for example, to write the log to the disk. Ravel tries to limit the interference to ensure the replayed execution is faithful to the recorded one. For example, it loads the BT engine in an unused memory area, and asks the kernel to allocate the code cache in another unused area. Since the engine is separated from the code cache [2], Ravel can tell whether a syscall is made by the engine or the program itself, and makes the replay decision accordingly. Note that BT engines often make direct syscalls rather than calling libc functions to avoid disrupting the translated process because many libc functions are non-reentrantable or thread-unsafe.

Ravel records the complete execution history of the target process. Even though replaying is often much faster than recording [41], it may still take a long time to replay a long-running process, such as a web server. To address that, we can take periodic snapshots of the process and start replaying from the most recent snapshots if the recorded history is too long. We should continue searching backwards for vulnerabilities until a vulnerability is located. This may introduce false negatives if there are multiple exploited vulnerabilities because of the missing/partial def-use relations. Our prototype does not support this optimization so far.

2.4 Pinpointing Vulnerabilities

Vulnerability locator aims at pinpointing the targeted vulnerabilities from a recorded execution that is known to contain attacks. It is based on the key observation that memory exploits often change the data flow. As such, it first uses a data-flow analysis to locate the rough locations of the vulnerability and further refines them with specific analyses targeting common types of memory vulnerabilities. Ravel is designed as an extensible framework so that analyses for less common types of vulnerabilities can be added later.

2.4.1 Data-Flow Analysis

Ravel calculates the probable locations of an exploited vulnerability using the data-flow analysis. We define a program's data flow as the def-use relations between instructions [11]. Specifically, an instruction defines a memory address if it writes to that address, and an instruction uses a memory address if it reads from that address. If two instructions define and use the same address respectively, they form a def-use relation. To detect data-flow anomalies, Ravel computes a data-flow graph (DFG) for the program beforehand using dynamic analysis. During the replay, it instruments the process to capture a detailed execution log, including all the run-time memory accesses. It then extracts from this log the data-flow of the program under attack. If an actual def-use relation is not in the pre-computed DFG, we consider both instructions of this def-use relation as a candidate for the vulnerability. However, it is unclear which instruction is the one.

Ravel uses heuristics to help determine whether the "def" or the "use" more likely marks the vulnerability. *First*, if multiple def-use relations are introduced by one of these instructions, that instruction more likely is the vulnerability. For example, in a buffer overflow, the def instruction may overwrite a large block of memory used later by several instructions. *Second*, if the data is used by a syscall that sends data outside (e.g., `send`, `sendmsg`, `write`), the vulnerability is likely related to the use (i.e., an information leak). *Third*, if the accessed memory contains control data, the vulnerability likely lies in the def instruction. We can identify control data if it is read by instruction fetching (e.g., a `return` instruction fetches its return address from the stack), or if it is subsequently used by an indirect branch instruction. If none of the above heuristics applies, Ravel reports both instructions as a viable candidate.

[2]In dynamic BT, the translated code executes from the code cache, instead of the original code section.

Ravel's data-flow analysis is performed at the instruction level. For a logged syscall, Ravel understands its semantics and can identify the memory regions it reads from and writes to. From this perspective, a syscall can be treated as a pseudo instruction with an extended define and use sets. Moreover, if an identified instruction lies inside a library, we trace back from that instruction until we reach its call site. A call to a library function is normally encoded as the call to a PLT (procedure linkage table [3]) entry. This step is necessary otherwise many identified vulnerabilities would be erroneously attributed to library functions. For example, buffer overflows are often caused by incorrect use of library functions like `strcpy` and `memcpy`.

Ravel's data-flow analysis can cover lots of memory vulnerabilities as most memory-corrupting exploits disturb the program's data flow. For example, it can locate both memory vulnerabilities in Figure 2. Specifically, line 7 contains a buffer overflow. If exploited, it defines the overflowed data on the stack, including the return address pushed by the caller of `process_request`. When the return address is fetched at line 10, an extra def-use relation is detected between these lines. Since this use is an instruction fetch, Ravel reports that the vulnerability lies in line 7. Line 9 contains an information leak. If exploited, it reads beyond `buffer_to_send`, creating multiple def-use relations with the same use. Ravel thus reports line 9 as the potential vulnerability. However, as demonstrated by Figure 2, the data-flow analysis is not the final solution. The real vulnerability could hide in somewhere else. To this end, Ravel tries to further refine the results with the following analyses.

2.4.2 Integer Errors

Integer errors usually are not exploited alone but followed by other exploits. For efficiency, Ravel focuses on common integer errors associated with buffer access violations. Specifically, it checks for common integer errors if a reported vulnerability involves block memory operations as these operations are often conditioned by a length parameter. Such vulnerabilities include calls to popular library functions that are frequently associated with buffer overflows (e.g., `memmove`, `memcpy`, `strncpy`, `strncat`, `strlcpy`, and their many variants) and I/O functions or syscalls (e.g., `recv`, `recvfrom`, and `read`). However, some programs may use their own block data copy/move functions rather than the standard `libc` functions. These functions often employ the repeated string instructions of x86 (i.e., instructions like `MOVS/STOS/CMPS/LODS/SCAS` prefixed by `REP/REPE/REPNE/REPNZ/REPZ`). In this case, register `RCX` contains the data size, and register `DS:RSI` and `ES:RDI` contain the source and destination addresses, respectively. To locate integer errors, we search from the reported vulnerability backward.

There are several types of integer errors, such as assignment truncation, integer overflow/underflow, and signedness error. An assignment truncation happens when an integer is assigned from a longer type to a shorter type. This is usually done by simply discarding the extra most significant bits. An integer overflow/underflow happens when the result of an integer arithmetic exceeds the valid range of the target register. A signedness error happens when an integer is converted between a signed type and an unsigned type. The code in Figure 2 contains a signedness error at line 6. Note that regular C/C++ programs may contain both benign and harmful integer errors [24]. However, false positives will not be a big issue for Ravel since all its detected integer errors are related to a reported vulnerability and thus are more likely to be true positives.

Detecting assignment truncations is relatively simple because x86's instruction encoding specifies the width of memory or register accesses. Meanwhile, integer overflows/underflows can be detected by checking the `RFLAGS` register, which contains various bits for arithmetic instructions. However, signedness errors are more challenging to identify because many integer instructions do not encode the signs of their operands. For example, signed and unsigned integer additions/subtractions are essentially the same with the two's complement data format. To address that, we collect hints of the signs for those operands from other instructions that access them. Many instructions do carry sign information. For example, the `JG`, `JGE`, `JL`, and `JLE` instructions select a branch based on the signed comparison, while `JA`, `JAE`, `JB`, and `JBE` instructions are based on the unsigned comparison. Conditional move instructions, such as `CMOVG` and `CMOVA`, also carry the sign information. Modern compilers tend to use conditional moves for better performance. Some arithmetic instructions also carry the sign information, such as `SAR/SHR` and `IDIV/DIV`. A second source of the sign information comes from the reported vulnerability itself. For example, functions like `memcpy` provide a clear definition of their parameter types. If there are conflicts in the collected hints of signs, a signedness error is highly likely and will be reported.

```
1 uint8_t *buffer = new uint8_t[size+chunk_size];

2 if (size > 0) {
3     memcpy(buffer, data, size);
4 }
```

Figure 4: Code snippet of CVE-2015-3864 in Android. There is an integer overflow in line 1, leading to the buffer overflow on line 3.

Ravel can detect the signedness error in Figure 2. Specifically, the data-flow analysis locates the vulnerability in line 7 (or line 9 depending on the attack), which specifies that its parameter `size` is unsigned. Searching backwards, Ravel finds that `size` was assigned from a signed number (`min` uses a signed comparison instruction). This conflict in `size`'s signs allows Ravel to identify this integer error. Figure 4 shows an example of integer overflows. The code is related to CVE-2015-3864 [17], an integer overflow in Android's built-in Stagefright media library. If `size + chunk_size` overflows, the allocated buffer may be smaller than the data length, leading to the buffer overflow in line 3. Ravel can detect this integer overflow by checking the RFLAGS register during the replay.

2.4.3 Use-After-Frees and Double-Frees

Ravel instruments the memory allocation/free functions to keep track of the memory life-time. This information can confirm buffer overflows (the buffer size vs. the data size) and identify use-after-free and double-free flaws. Use-after-free and double-free have become popular attack vectors in recent years. Even though the data-flow analysis can discover anomalies in the data flow caused by them, it does not have enough information to correctly identify them. This insufficiency can be addressed by the memory life-time information. For example, a vulnerability can be categorized as use-after-free if a block of the freed memory is accessed again. Note that some programs use their own memory management functions rather than the standard libc or C++ functions. This issue can be addressed through source code annotation or with heuristics.

2.4.4 Race Conditions

With the ubiquitous deployment of exploit mitigation techniques like DEP and ASLR, race conditions have become a more popular attack vector. As previously mentioned, Ravel's replayed execution may deviate from the recorded one if the program has race conditions. For example, they may have different syscall sequences, or the replayed execution crashes but the recorded one does not.

[3]PLT/GOT is the structure to support dynamic linking [44].

When that happens, Ravel runs an existing algorithm [2] to detect race conditions during the replay. Specifically, it checks whether two potentially racing operations (e.g., two threads write to the same variable) have a happens-before relation, i.e., one operation is guaranteed to happen before the other. Such a relation can be established if these operations are protected by locks or ordered through inter-process communications (e.g., pipes). We plan to add the capability to root-cause race conditions by identifying common data racing patterns [37].

2.5 Prototype Efforts

We have implemented a prototype of Ravel based on the FreeBSD release 10.2. The R&R system is implemented from scratch in the kernel with a small user-space utility to control recording and replaying. We added about 3.9K SLOC (source lines of code) to the kernel, and the utility consists of about 300 SLOC. Vulnerability locator is based on the open-source Valgrind [47] instrumentation framework. We made some changes to the Valgrind framework itself so that it could be used in the replay (Section 2.3.2). In addition, Valgrind's system call wrappers for FreeBSD are incomplete. We wrote our own and contributed it back to the project. We added about 2.2K SLOC to Valgrind in total. The replayer captures the whole execution trace of the replayed program, specifically, the executed instructions and the addresses and sizes of their memory accesses. Based on the execution trace, we implemented Ravel's data-flow analysis, integer error detection, and use-after-free and double-free detection, and integrated Valgrind's existing race condition detection. Most of these analyses work at the byte granularity except the integer error detection, which works according to the instructions' operand sizes. As mentioned before, Ravel works on program binaries directly. After locating a vulnerability, we revert it back to the source code if the program contains the debugging symbols.

3. EVALUATION

In this section, we first evaluate the effectiveness of Ravel against common memory-based vulnerabilities and then measure the performance overhead caused by Ravel.

3.1 Effectiveness

To evaluate the effectiveness of Ravel, we first analyze how Ravel can handle common memory vulnerabilities, such as buffer overflows, integer errors, information leaks, and format string vulnerabilities. We then describe our experiments with a variety of vulnerabilities, including two high-impact real-world ones.

By design, Ravel can detect any attacks that change the program's run-time data flow. However, the data-flow analysis itself often cannot provide the precise locations of the exploited vulnerabilities. To address that, Ravel further refines the results with vulnerability-specific analyses. There are many types of common memory vulnerabilities. In the following, we discuss how Ravel can handle some of them, starting with the most common one, buffer overflows.

Buffer overflows: a buffer overflow, or a buffer overrun, happens when a program writes more data into a buffer than it can hold, overwriting the adjacent data. A typical example is to use functions like `strcpy` that do not check the buffer size to copy untrusted data. Buffer overflows can often lead to arbitrary code execution and denial-of-services. Ravel can locate buffer overflows if the overwritten data are used after the attack: when a piece of data is read, a new def-use relation is introduced between the vulnerability and the reader. In addition, Ravel could tell that the def is likely the vulnerability if there are multiple new uses with the same def. An example is shown in Figure 2 line 7 in which a new def-use relation is introduced between line 7 and 10.

Integer errors: integer errors include a number of flaws related to integer operations such as arithmetic, type casting, truncation, and extension. For example, an integer overflow happens when the result of an integer arithmetic exceeds the valid range of the destination type. Integer errors are often not exploited alone but instead with other vulnerabilities, such as buffer overflows. To detect integer errors, Ravel first locates the symptomatic vulnerability and then searches for possible integer errors if that vulnerability takes integer parameters. In Figure 2, after locating the buffer overflow in line 7, Ravel continues to search for integer errors because `recv`'s `size` parameter is and integer and discovers the integer signedness error in Line 6.

Information leaks: an information leak happens when a program inadvertently leaks data to unauthorized parties that may help them obtain sensitive information or launch further attacks. For example, an attacker often exploits information leaks to de-randomize the victim process' address space before launching return-oriented programming attacks. Information leaks can also disclose confidential information to attackers. A recent high-profile example is the Heartbleed flaw in the OpenSSL library. Heartbleed can be exploited to leak the server's memory to the attacker, $64KB$ at a time. This eventually allows the attacker to steal the server's private key. Ravel can precisely locate information leaks: an information leak reads more data than it should. This creates additional def-use relations between the writers of that data and the vulnerability. Accordingly, Ravel can tell that the use likely is the vulnerability since there are multiple defs with the same use. Afterwards, Ravel tries to identify integer errors. In Figure 2, new def-use relations are introduced between line 9 and the writers of the data adjacent to `buffer_to_send` (not shown in the figure).

Use-after-frees: use-after-frees are another common type of memory vulnerabilities, in which a program erroneously references the memory that has been previously freed. Depending on its nature, a use-after-free may allow an attacker to crash the program, corrupt data, or even execute the injected code. In a typical scenario to exploit a use-after-free, the attacker tries to allocate an object under his/her control immediately after the vulnerable memory is freed. The memory allocator likely assigns the just-freed memory to this object, giving the attacker full control over the to-be-reused memory. If the reused memory originally contains a data pointer, the attacker could exploit it to read or write arbitrary data. Likewise, if it contains a code pointer, the attacker could exploit it to hijack the control flow. Ravel can locate a use-after-free if the attacker-controlled object is different from the vulnerable object (this is often the case otherwise the attacker can simply misuse the object under his control). Consequently, they are accessed by different instructions, and new def-use relations are created from the writers of the attacker-controlled object to the readers of the vulnerable object. Ravel also keeps track of the data lifetime to facilitate the detection of use-after-frees and double-frees.

Format string vulnerabilities: a format string vulnerability occurs when a function, such as `printf`, accepts an attacker-controlled format string. The format string decides how the function interprets its following parameters. By manipulating format directives, an attacker can read data from the stack, corrupt memory, and even execute arbitrary code. Ravel can pinpoint format string vulnerabilities. For example, a new def-use relation will be introduced between the format function and its caller if the vulnerability is exploited to read the return address. Format string vulnerabilities are becoming less common nowadays because it is relatively easy for compilers to automatically detect them. For example, `gcc` allows

```
1  typedef struct {
2      ...
3      off_t content_length_n;
4      ...
5  } ngx_http_headers_in_t;
6  ...
7  u_char buffer[NGX_HTTP_DISCARD_BUFFER_SIZE];
8  ...
9  size = (size_t) ngx_min(r->headers_in.
        content_length_n, NGX_HTTP_DISCARD_BUFFER_SIZE
        );
10 n=r->connection->recv(r->connection, buffer, size);
```

Figure 5: Code sketch of CVE-2013-2028 in NGINX. An integer signedness error at line 9 leads to a buffer overflow at line 10.

a program to annotate its own format functions with the `format` compiler directive. This feature is extensively used by the Linux kernel to protect its debugging and logging functions.

So far, we have discussed Ravel's effectiveness in locating common memory vulnerabilities. We also experimented with a number of real-world and synthetic vulnerabilities and attacks. In the following, we will give the details of several such experiments.

3.1.1 CVE-2013-2028 of NGINX

NGINX is a popular open-source web server. It powers some of the most popular web sites on the Internet, such as Netflix, Hulu, CloudFlare, and GitHub [49]. Vulnerabilities in NGINX consequently have very high impacts.

Figure 5 shows the CVE-2013-2028 vulnerability in NGINX (version `1.3.9` to `1.4.0`). This flaw lies in NGINX's faulty handling of chunked transfer-encoding, a standard feature of HTTP 1.1. This feature allows data to be sent in a series of chunks. It replaces the regular Content-Length HTTP header with the "Transfer-Encoding: chunked" header. Because it allows data to be transferred piecemeal without knowing their total length, the chunked encoding is particularly useful for dynamically generated data. In this encoding, the length of each chunk is prefixed to the actual data in the chunk. A vulnerable NGINX server parses the chunk length and stores it in `r->headers_in.content_length_n`. The length is then compared to the buffer size in an effort to prevent buffer overflows. Unfortunately, there is a signedness error at line 9. Specifically, `content_length_n` is a signed integer. If the attacker makes it negative, `ngx_min` returns it without any change. The length is then casted to an unsigned integer (`size`). This vulnerability could be exploited to overflow the buffer on the stack (`buffer` defined at line 7) at line 10.

Interestingly, this vulnerability cannot be exploited as is on the FreeBSD system because the FreeBSD kernel prevents system calls like `recv` from accepting unreasonably-large size parameters. This is essentially an ad-hoc syscall parameter validation. In spite of that, it is a rather effective defense against similar attacks without requiring any change to user programs. With this protection, the buffer overflow at line 10 is foiled by the kernel, i.e., the kernel has located this vulnerability. As such, we can omit the data-flow analysis and directly start other analyses. Ravel reports the signedness error at line 9. If we remove the kernel's protection, this vulnerability becomes exploitable. Note that even though the `size` parameter to `recv` is really large, the attack can control how much data to be returned by `recv` because `recv` returns the existing data cached in the socket without waiting for the full requested size. In this experiment, we launched a return-oriented programming (ROP [54]) attack against the server, similar to Blind ROP [7]. Exceptions caused by the attack allowed Ravel to detect the attack and identify the buffer overflow at line 10. Ravel further traced it back to the signedness error at line 9 based on the conflicts in the

```
1  /* ssl/d1_both.c */

2  int dtls1_process_heartbeat(SSL *s)
3  {
4      unsigned char *p=&s->s3->rrec.data[0], *pl;
5      unsigned short hbtype;
6      unsigned int payload;
7      unsigned int padding = 16;
8      ...
9      /* Read type and payload length first */
10     hbtype = *p++;
11     n2s(p, payload);
12     ...
13     pl = p;
14     if (hbtype == TLS1_HB_REQUEST) {
15         unsigned char *buffer, *bp;
16         ...
17         int r;
18         ...
19         buffer = OPENSSL_malloc(1 + 2 + payload+
                padding);
20         bp = buffer;
21         /* Enter response type, length and copy
                payload */
22         *bp++ = TLS1_HB_RESPONSE;
23         s2n(payload, bp);
24         memcpy(bp, pl, payload);
25         bp += payload;
26         /* Random padding */
27         RAND_pseudo_bytes(bp, padding);
28         r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT,
                buffer, 3 + payload + padding);
29         ...
30     }
31     ...
32 }
```

Figure 6: Code sketch of CVE-2014-0160 (aka. Heartbleed). The attacker controls `payload`. `Memcpy` may copy a large amount of extra data to `buffer`, and send it back through `dtls1_write_bytes`.

hinted signs of `size`: unsigned at line 10 (deduced from `recv`), and signed at line 9. In particular, `ngx_min` is compiled into a signed comparison followed by a conditional move instruction (`cmovg`) .

3.1.2 CVE-2014-0160 (Heartbleed)

CVE-2014-0160, commonly known as Heartbleed, is an information leak in the popular OpenSSL library. OpenSSL is a ubiquitous open-source TLS/SSL and cryptography library. It is shipped in most Unix-like systems (e.g., Linux and BSDs) and available for other systems. It is also embedded in lots of devices like routers. The impacts of this flaw is serious and far-reaching.

Heartbleed is a flaw in OpenSSL's handling of the heart-beat extension, which essentially is an echo protocol (similar to `ping` in IP) to ensure the liveliness of an SSL connection. Heartbleed allows an attacker to steal the victim's memory up to 64KB at a time. It has been demonstrated that critical data, such as the private key for TLS/SSL, could be leaked by this flaw. Figure 6 shows the code sketch of this flaw. Specifically, the `payload` field of the heartbeat request packet specifies how many bytes of the request data should be sent back to the requester. This field is extracted from the request packet at line 11. A response buffer is allocated at line 19 and the response is assembled in it by line 22 to 27. Particularly, line 24 copies the data from the request packet to the buffer. Finally, the response is sent to the requester at line 28. This bug is caused by the missing check that the request payload has more bytes than `payload`. Because the `payload` field is 16 bits, at most $65,535$ bytes of the data can be exfiltrated each time.

In our experiment, we ran OpenSSL 1.0.1 with NGINX to serve HTTPS requests. We kept exploiting this bug with different combinations of requests in order to obtain the server's private key.

Program Name	Vulnerability Type	Pinpointed?
BitBlaster	Null Pointer Derefernece	Yes
CGC_Planet_Markup_Language_Parser	Heap Overflow	Yes
	NULL Pointer Dereference	Yes
	StackOverflow	Yes
CGC_Board	Heap Overflow	Yes
CGC_Symbol_Viewer_CSV	Integer Overflow	Yes
CGC_Video_Format_Parser_and_Viewer	Heap Overflow	Yes
simple_integer_calculator	Heap Overflow	Yes
	Integer Overflow	Yes
	NULL Pointer Dereference	Yes
	Out-of-bounds Read	Yes
Diary_Parser	Null Pointer Dereference	Yes
	Out-of-bounds Read	Yes
	Stack Overflow	Yes
electronictrading	Heap Overflow	Yes
	Integer Overflow	Yes
	Untrusted Pointer Dereference	Yes
	Use After Free	Yes
Enslavednode_chat	Heap Overflow	Yes
Kaprica_Script_Interpreter	Arbitrary Format String	Yes
	NULL Pointer Dereference	Yes
KTY_Pretty_Printer	Double Free	Yes
	Stack Overflow	Yes

Table 1: Summary of the evaluation results on a number of DARPA CGC programs.

Moreover, the server was configured to automatically restart if it crashed due to invalid memory reads caused by the attack. After catching an exception, Ravel replayed the attack and discovered extra def-use relations from other instructions to `memcpy` at line 24. We concluded that the vulnerability lied at line 24 because those relations had a common use. Ravel further checked for integer errors. None was found.

3.1.3 *Experiments with CGC Challenges*

To evaluate Ravel's effectiveness on a more diverse set of vulnerabilities, we used the sample challenges from DARPA's Cyber Grand Challenge (CGC) [18], a competition to design better Cyber Reasoning Systems that can automatically identify software flaws and scan for affected hosts in the network. Challenges in CGC are designed to represent a variety of software vulnerabilities. Instead of contrived simple test cases, they approximate real software vulnerabilities with enough complexity, ideal for stressing vulnerability discovery and defense systems. Each challenge comes with a set of proof-of-vulnerability (POV) inputs that can trigger these vulnerabilities. The record and replay of CGC programs are simple since they only use a very small number of system calls. In our experiments, we used Ravel to pinpoint these vulnerabilities. The results are summarized in Table 1. It shows that Ravel can help developers locate a variety of types of vulnerabilities. In the following, we give details for one of the challenges (CNMP).

The CNMP (Chuck Norris Management Protocol) challenge models a message management system, in which users can add, list, count, and show messages (jokes, to be more specific). The related vulnerable functions are listed in Figure 7. `insert_joke` inserts a message into the system's database. If the message's length exceeds a threshold (line 7), the system logs this message using `syslog` (line 10) and returns an error code. Inside the `syslog` function, its argument `format` is passed to `vsnprintf` (line 21), a string-formatting function. Since `joke_str` is controlled by the attacker, a format-string vulnerability can be triggered by passing a crafted string.

In this experiment, we added a long message with format specifiers to the system database. The program crashed due to its `vsnprintf` implementation. Ravel replayed the program and reported that over-read happened inside function `vsnprintf`, where extra def-use relations were introduced. Since this function takes a format string as its input, it is easy to figure out the vulnerability by looking at the recorded function arguments (a format string). A developer can fix this vulnerability by using "%s" as the format string in Line 10.

```
 1 // add joke to joke_db.
 2 int insert_joke(jokedb_struct *jokedb, const char *
      joke_str) {
 3     // return error (-1) if jokedb is already full.
 4     if (jokedb->count >= MAX_JOKES) {
 5         return -1;
 6     // return error (-2) if joke_str is too long.
 7     } else if (strlen(joke_str) >=
           MAX_JOKE_STRING_LEN - 1) {
 8         if (LOGLEVEL >= LOG_INFO) {
 9             syslog(LOG_ERROR, "Joke was too long
                   -->\n");
10             syslog(LOG_ERROR, joke_str);
11         }
12         return -2;
13     } else {
14         ...
15     }
16 }

17 int syslog(int priority, const char *format, ...) {
18     ...
19     // process format string
20     // and write it to log_entry buffer
21     log_entry_len += vsnprintf(log_entry_idx,
           MAX_SYSLOG_LEN - log_entry_len, format,
           args);
22     ...
23     return 0;
24 }
```

Figure 7: Code sketch of vulnerability-related functions in CNMP. `syslog` takes the user-controlled `joke_str`, and passes it as a format-string argument to `vsnprintf`.

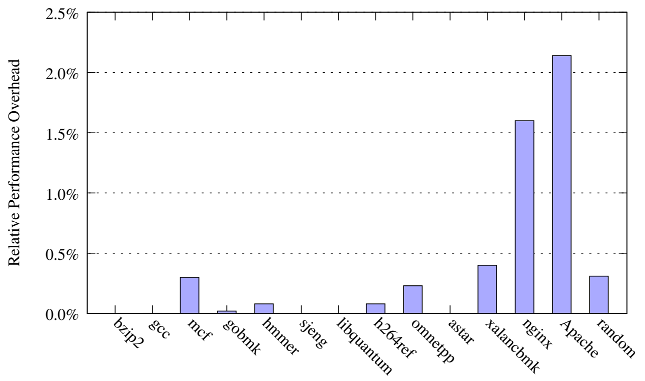

Figure 8: Performance overhead of Ravel's online components relative to the original FreeBSD system.

3.2 Performance

To make the most use of Ravel, it should be deployed in production systems to catch in-the-wild, zero-day vulnerabilities. This imposes a strict performance requirement on both its design and implementation. However, locating vulnerabilities requires time-consuming analyses of the program's memory access patterns. To address that, we use record & replay (R&R) to separate the system into an online component and an offline component. The former records the program's execution and detects ongoing attacks; while the latter replays the execution to locate vulnerabilities. As such, the performance of the online component is more critical than the offline component and is the focus of our performance evaluation. The online overhead can mainly be attributed to the attack detection and the recording. Our prototype employs two lightweight attack detection mechanisms. As such, most of the overhead comes from the recording.

We measured the performance of Ravel with a set of standard benchmarks (SPEC CPU 2006 [4]) and several real-world applications. All the experiments were conducted on an Intel Core i7 computer with 16 GB of memory. The OS was based on FreeBSD release 10.2 for x86-64. All the test applications were installed directly from FreeBSD's software repository. These applications include two popular web servers, NGINX and Apache. We used ApacheBench to send 5×10^6 requests to each server with a concurrency of 10. To diversify the tests, Apache was configured to work in the `worker` mode with multi-threading enabled; while NGINX was configured to use `poll`, instead of `kqueue` as it is in Apache, for connection processing. The third application we tested was `dd`. Specifically, we used it to read 10MB of data from the kernel's random number pool (`/dev/random`) and compressed the data with `lzma`. The command line was `dd if=/dev/random bs=1k count=10000 | lzma > /dev/null`. As mentioned before, random numbers are a source of non-determinism and are always recorded.

The results are illustrated in Figure 8. It shows that the performance overhead of Ravel for most CPU-intensive benchmarks is negligible. For I/O intensive ones like the web servers, the overhead is also rather small at about 2%. This is expected as the recording mostly happens when there is a syscall, a relatively infrequent event given the speed of today's computers. Our optimization of the storage consumption for R&R allows Ravel to avoid recording the data that can be recovered from the environment, such as HTML files. However, Ravel has to record the data read from the random pool because they cannot be reproduced.

[4]SPEC CPU2006 has dropped the support to FreeBSD for a long time. There are many compatibility issues that prevent some benchmarks from compiling despite all the compilers we tried (including the official `Clang/LLVM` compiler and several versions of `gcc`).

4. DISCUSSION

In this section, we discuss potential ways to improve the design and implementation of Ravel and the future work. *First*, Ravel focuses on pinpointing vulnerabilities from a recorded attack. The attack detector plays an important role in the overall effectiveness of Ravel. Ravel has been designed not to rely on details of the detector – its only input is a recorded execution history that is known to contain some attacks. Therefore, it is possible to integrate a wide range of attack detection techniques. For example, recent advances in control-flow integrity [52, 61, 62] make it a practical choice as the attack detector. Effective and low-overhead attack detection for data-only and other emerging attacks is still an ongoing research topic.

Second, Ravel uses R&R to detach the time-consuming vulnerability locator from the production system and make attacks reproducible for offline analyses. Ravel's R&R is an always-on kernel-based R&R system for processes. This imposes strict requirements in its performance and storage overhead. In the future, we plan to integrate more techniques in eidetic systems [23] to optimize the storage for long-running processes. As expected, this will increase the performance overhead (eidetic systems have about 8% of overhead). Another challenge is how to reduce the replay time for long-running processes even though replaying is much faster than recording (because most syscalls are not re-executed). To address that, we may periodically checkpoint the process and start replaying at the nearest checkpoints. The challenge is then how to locate vulnerabilities from a potentially incomplete execution history.

Third, by design, Ravel can only locate attacks that change the data flow. It is ineffective against attacks that do not do so. For example, SQL injections can be leveraged to execute malicious SQL queries without exploiting any flaws in the SQL server, and vulnerabilities in Java programs can be exploited without compromising the Java virtual machine. Other examples include attacks that misuse the benign/obsolete program features for malicious purposes, such as Shellshock [65]. Moreover, the design of Ravel may cause imprecision in the analyses. First, we use dynamic analysis to generate the program's data-flow graph (DFG). The incompleteness in the DFG may lead to false positives. We plan to explore methods to increase the code coverage of the dynamic analysis to build better DFGs [58]. Second, Ravel relies on heuristics to refine the locations of vulnerabilities, such as the signs of integers. Heuristics is by nature imprecise and may introduce both false positives and false negatives. Third, Ravel keeps track of memory lifetime to aid the detection of double-frees and use-after-frees. This method is less effective for programs that use custom memory allocators. We could use (more) heuristics to automatically detect custom memory allocators.

5. RELATED WORK

In this section, we present and compare with the related work regarding attacks and defenses of memory-based vulnerabilities.

Attack detection and exploit mitigation: the first category of the related work is a long stream of research in the attack detection and exploit mitigation [1, 3, 4, 6, 11, 21, 22, 29, 33, 51, 10, 63, 64, 66, 67] (a recent survey paper provides a comprehensive overview of these systems [59]). For example, $W \oplus X$ [22] and ASLR [60] are two widely adopted exploit mitigation techniques. $W \oplus X$ prevents executing the data and modifying the code. However, it can be bypassed by reusing the existing code, such as return-to-libc and return-orient programming (ROP) [8, 54]. Many systems have been proposed to defeat ROP, say, by diversifying the program's code [42]. For example, ASLR randomizes the process' address space layout to make it harder for attackers to locate the reusable code. ASLR is susceptible to information leaks [56]. Many improvements to ASLR have been proposed to address this problem by making ASLR more fine-grained [33, 51, 64], preventing code from being read [4, 15], and real-time re-randomization [6, 14]. Nevertheless, $W \oplus X$ and ASLR have significantly raised the bar for reliable exploits. These defenses can be integrated with Ravel to enable the defense in depth. They will also improve Ravel's attack detection since they make exploits crash more often.

From another perspective, control-flow integrity (CFI [1]) and data-flow integrity (DFI [11]) provide a comprehensive protection against control-flow and data-flow attacks, respectively. They enforce the security policy that run-time control flow/data flow must follow the program's control-flow/data-flow graph. CFI is an effective defense against most control-flow attacks. DFI has an even broader coverage because it can also detect data-only attacks. These two techniques have inspired a lot of related systems, including Ravel [21, 29, 57, 66, 67]. One of their focuses is to minimize the performance overhead so that they can be practically deployed [57, 66]. For example, Kenali enforces DFI for the kernel's access control system [57]. It automatically infers the critical data that need protection and enforces DFI for that data. CFI as a generic attack detection technique can be integrated into Ravel. Ravel's architecture can be easily extended with all kinds of attack detection techniques. Ravel's data-flow analysis extends DFI with analyses to locate and refine underlying vulnerabilities. As we have demonstrated, a data-flow violation does not always coincide with the exploited vulnerabilities. Moreover, full DFI enforcement is still prohibitively expensive [11]. Ravel's use of R&R detaches the (high-overhead) vulnerability locator from production systems. WIT (Write Integrity Testing) is another effective defense against memory errors [3]. It enforces a policy in which each instruction can only write to the set of statically-determined, authorized objects. WIT reduces its overhead by checking only memory writes but not reads. Consequently, WIT cannot detect read-only vulnerabilities such as information leaks. Ravel checks both memory reads and writes for vulnerabilities

Data-only attacks: as control-flow hijacking becomes more challenging, more attention is paid to data-only attacks (or non-control-data attacks) [13, 35, 36]. Data-only attacks do not change the control flow. They instead manipulate sensitive run-time data to indirectly control the program's execution, escalate privileges, leak information, etc. Recently, data-oriented programming (DOP) [36] demonstrates that attackers can systemically construct expressive non-control-data exploits, and a large percentage of the evaluated real-world programs have gadgets to simulate arbitrary computations. Effective, low-overhead detection of data-only attacks is a challenging problem. DFI can detect many but not all data-only attacks (i.e., it cannot detect attacks that do not change the def-use relations). Likewise, Ravel can also locate many but not all data-only vulnerabilities.

Vulnerability/bug discovery: There are also many efforts to discover vulnerabilities through dynamic and static analysis [5, 25, 55, 58]. For example, fuzz testing is a popular, practical approach to discover software vulnerabilities. It tries to crash a program by feeding it random inputs. However, fuzz testing often ends with a poor code coverage. To address that, Driller uses concolic execution to guide the fuzzer when it stops [58]. To evaluate vulnerability discovery tools, LAVA [25] proposes a dynamic taint-analysis based approach to generate large ground-truth vulnerability corpora on demand. Gist is a tool to diagnose program failures (i.e., crashes) [37]. Specifically, it traces the program execution with Intel's processor tracing technology and combines static program slicing and dynamic analysis to find root causes of program failures. Ravel shares the same vision as Gist, but it takes a very different approach because of its different focus — Gist focuses on solving program failures, while Ravel focuses on locating vulnerabilities. Many vulnerabilities such as buffer overflows and information leaks can be exploited without causing program failures, rendering Gist ineffective in locating vulnerabilities.

Record & replay: the last category of related work is record & replay (R&R) [23, 31, 38, 41, 53]. In particular, Arnold is an always-on R&R system. It enables an interesting concept called the eidetic system in which the complete execution history of the system is kept and can be queried for information about the past execution [23]. We adopt some techniques of Arnold to reduce Ravel's performance and storage overhead. Ravel's R&R is different from other R&R systems with its instrumented replay. R&R has a variety of interesting usages including software debugging and intrusion detection. For example, BackTracker can reconstruct a past intrusion by building an attack dependence graph backwards [38]. Ravel also relies on R&R to reproduce an attack, but it aims at locating vulnerabilities. BackTracker works on the high-level objects such as processes and files; while Ravel works on the low-level memory accesses with different analyses.

6. SUMMARY

We have presented the design and evaluation of Ravel, a practical system to pinpoint exploited vulnerabilities. Specifically, it records the execution history of a production program and simultaneously monitors its execution for attacks. If an attack is detected, the execution history is replayed with instrumentation to locate the exploited vulnerabilities. With its data-flow and other analyses, Ravel can pinpoint many different types of memory vulnerabilities, such as buffer (heap) overflows, integer errors, and information leaks. Our evaluation shows that Ravel is effective and incurs low performance overhead.

7. ACKNOWLEDGMENTS

We would like to thank our shepherd Mathias Payer and the anonymous reviewers for their insightful comments that greatly helped improve the presentation of this paper. This work was supported in part by the US National Science Foundation (NSF) under Grant 1453020. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

8. REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, November 2005.

[2] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. Netzer. Detecting Data Races on Weak Memory Systems. *ACM SIGARCH Computer Architecture News*, 19(3):234–243, 1991.

[3] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing Memory Error Exploits with WIT. In *Proceedings of the 29th IEEE Symposium on Security and Privacy*, May 2008.

[4] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, 2014.

[5] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier. A Taint Based Approach for Smart Fuzzing. In *Proceedings of the IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 818–825. IEEE, 2012.

[6] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. Timely Rerandomization for Mitigating Memory Disclosures. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, pages 268–279. ACM, 2015.

[7] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. Hacking Blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, pages 227–242. IEEE, 2014.

[8] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, October 2008.

[9] N. Burow, S. A. Carr, S. Brunthaler, M. Payer, J. Nash, P. Larsen, and M. Franz. Control-flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys*, 2017.

[10] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proceedings of the 24th USENIX Security Symposium*, volume 14, pages 28–38, 2015.

[11] M. Castro, M. Costa, and T. Harris. Securing Software by Enforcing Data-flow Integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, 2006.

[12] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 378–387. IEEE, 2005.

[13] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.

[14] Y. Chen, Z. Wang, D. Whalley, and L. Lu. Remix: On-demand Live Randomization. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 50–61, New Orelans, LA, Mar 2016. ACM.

[15] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *36th IEEE Symposium on Security and Privacy (Oakland)*, May 2015.

[16] CVE-2013-2028. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2028.

[17] CVE-2015-3864. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3864.

[18] DARPA. Cyber Grand Challenge. https://cgc.darpa.mil.

[19] Memory Protection Technologies. http://technet.microsoft.com/en-us/library/bb457155.aspx.

[20] x86 NX support. http://lwn.net/Articles/87814/.

[21] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *Proceedings of the 23rd USENIX Conference on Security*, 2014.

[22] Data Execution Prevention. http://en.wikipedia.org/wiki/Data_Execution_Prevention.

[23] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen. Eidetic Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, 2014.

[24] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding Integer Overflow in C/C++. *ACM Transactions on Software Engineering and Methodology*, 25(1):2, 2015.

[25] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. LAVA: Large-scale Automated Vulnerability Addition. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, May 2016.

[26] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing Near-optimal Malware Specifications from Suspicious Behaviors. In *Proceedings of the 31th IEEE Symposium on Security and Privacy*, pages 45–60. IEEE, 2010.

[27] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the 20th Annual Network and Distributed Systems Security Symposium*, February 2003.

[28] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Proceedings of the 10th Network and Distributed System Security Symposium*, 2003.

[29] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of Control: Overcoming Control-Flow Integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[30] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In *Proceedings of the 6th USENIX Security Symposium*, 1996.

[31] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An Application-level Kernel for Record and Replay. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.

[32] J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka. Verifying Information Flow Goals in Security-enhanced Linux. *Journal of Computer Security*, 13(1):115–134, 2005.

[33] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd My Gadgets Go? In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, pages 571–585. IEEE, 2012.

[34] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion Detection Using Sequences of System Calls. *Journal of computer security*, 6(3):151–180, 1998.

[35] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic Generation of Data-Oriented Exploits. In *Proceedings of the 24th USENIX Security Symposium*, pages 177–192, 2015.

[36] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, May

2016.

[37] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea. Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-production Failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 344–360. ACM, 2015.

[38] S. T. King and P. M. Chen. Backtracking Intrusions. In *Proceedings of the 2003 Symposium on Operating Systems Principles*, October 2003.

[39] A. P. Kosoresow and S. A. Hofmeyr. Intrusion Detection via System Call Traces. *IEEE software*, 14(5):35, 1997.

[40] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the Detection of Anomalous System Call Arguments. In *Proceedings of the 8th European Symposium on Research in Computer Security*, pages 326–343. Springer, 2003.

[41] O. Laadan, N. Viennot, and J. Nieh. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *Proceedings of the 2010 International Conference on Measurement and Modeling of Computer Systems*, 2010.

[42] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated Software Diversity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, pages 276–291. IEEE, 2014.

[43] W. Lee, S. J. Stolfo, et al. Data Mining Approaches for Intrusion Detection. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

[44] J. R. Levine. *Linkers and Loaders*. Morgan Kaufmann, San Francisco, CA, 1999.

[45] F. Maggi, M. Matteucci, and S. Zanero. Detecting Intrusions Through System Call Sequence and Argument Analysis. *IEEE Transactions on Dependable and Secure Computing*, 7(4):381–395, 2010.

[46] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel. Anomalous System Call Detection. *ACM Transactions on Information and System Security*, 9(1):61–93, 2006.

[47] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[48] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 12th Network and Distributed System Security Symposium*, Feburary 2005.

[49] NGINX. NGINX. https://www.nginx.com.

[50] S. Palahan, D. Babić, S. Chaudhuri, and D. Kifer. Extraction of Statistically Significant Malware Behaviors. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 69–78. ACM, 2013.

[51] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

[52] M. Payer, A. Barresi, and T. R. Gross. Fine-grained Control-flow Integrity through Binary Hardening. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 144–164. Springer, 2015.

[53] M. Ronsse and K. De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems*, 17(2), May 1999.

[54] 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*, October 2007.

[55] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. (State of) The Art of War: Offensive Techniques in Binary Analysis . In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, May 2016.

[56] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time Code Reuse: On the Effectiveness of Fine-grained Address Space Layout Randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588. IEEE, 2013.

[57] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In *Proceedings of the 23rd Network and Distributed System Security Symposium*, Feb 2016.

[58] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 23rd Network and Distributed System Security Symposium*, Feb 2016.

[59] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal War in Memory. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[60] P. Team. PaX Address Space Layout Randomization (ASLR), 2003.

[61] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical Context-sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 927–940. ACM, 2015.

[62] V. van der Veen, E. Göktas, 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 *Proceedings of the 37th IEEE Symposium on Security and Privacy*, pages 934–953. IEEE, 2016.

[63] X. Wang, Y. Chen, Z. Wang, Y. Qi, and Y. Zhou. SecPod: a Framework for Virtualization-based Security Systems. In *Proceedings of the 2015 USENIX Annual Technical Conference*, pages 347–360, 2015.

[64] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, 2012.

[65] Wikipedia. Shellshock (software bug). https://en.wikipedia.org/wiki/Shellshock_(software_bug).

[66] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, 2013.

[67] M. Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.