DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks

Alexey Smirnov Tzi-cker Chiueh
Computer Science Department
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
{alexey, chiueh}@cs.sunysb.edu

Abstract

Buffer overflow attacks are known to be the most common type of attacks that allow attackers to hijack a remote system by sending a specially crafted packet to a vulnerable network application running on it. A comprehensive defense strategy against such attacks should include (1) an attack detection component that determines the fact that a program is compromised and prevents the attack from further propagation, (2) an attack identification component that identifies attack packets so that one can block such packets in the future, and (3) an attack repair component that restores the compromised application's state to that before the attack and allows it to continue running normally. Over the last decade, a significant amount of research has been vested in the systems that can detect buffer overflow attacks either statically at compile time or dynamically at run time. However, not much effort is spent on automated attack packet identification or attack repair. In this paper we present a unified solution to the three problems mentioned above. We implemented this solution as a GCC compiler extension called DIRA *that transforms a program's source code so that the resulting program can automatically detect any buffer overflow attack against it, repair the memory damage left by the attack, and identify the actual attack packet(s). We used* DIRA *to compile several network applications with known vulnerabilities and tested* DIRA*'s effectiveness by attacking the transformed programs with publicly available exploit code. The* DIRA*-compiled programs were always able to detect the attacks, identify the attack packets and most often repair themselves to continue normal execution. The average run-time performance overhead for attack detection and attack repair/identification is 4% and 25% respectively.*

1. Introduction

A control-hijacking attack overwrites some data structures in a victim program that affect its control flow, and eventually hijacks the control of the program and possibly the underlying system. A data structure that can affect the control flow of a program is called a *control-sensitive* data structure, examples of which include return address, function pointer, global offset table/import table, C++ virtual functions table pointer, etc. Once an attacker grabs control of the victim program, she can invoke any operation to which the victim program's effective user is entitled. Control-hijacking attacks are considered the most dangerous type of attacks because they exploit software bugs directly without requiring any user actions, and because malicious computer worms use them as basic building blocks to propagate themselves from one machine to another.

Over the last decade, a significant amount of research has been invested in the detection of control-hijacking attacks. Some are based on program analysis techniques [38, 12, 19, 26, 32, 37] that statically determine whether a given program contains buffer overflow vulnerability. Others use program transformation techniques [8, 11, 10, 14, 36, 9] to convert applications into a form that can either detect control-hijacking attacks [8, 10, 14, 9] or prevent control-sensitive data structures from being modified at run time [11]. Still others develop operating system mechanisms that ensure that it is not possible to execute code injected into the victim program [34, 27]. Regardless of their approach, most if not all of these efforts could only determine whether a program is under a control-hijacking attack, but could not actively repair a victim program after it has been compromised. Typically, upon detecting an attack, they simply terminate the victim application, and restart another instance if necessary. While terminating a compromised application helps prevent further propagation of the attack, it may lead to a denial of service attack. For network applications with a substantial amount of state such as a DNS sever, it takes some time for them

	D	I	R
Stackguard [10], RAD[8]	+	-	-
Buttercup [29], Autograph [21]	-	+	-
Flashback [33], IGOR [13]	-	-	+
DIRA	+	+	+

Table 1. Previous work addressing problems of attack (D)etection, (I)dentification, and (R)epair.

to re-acquire the necessary state at start-up in order to provide the full service. For these applications, abrupt termination is not an acceptable attack recovery strategy. Moreover, because existing control-hijacking attack detection systems cannot prevent the same attacks from taking place again, vulnerable applications may be repeatedly victimized and re-started in the presence of recurring attacks as in the case of worms. In the mean time, these applications cannot render any useful service to their intended users.

To address the limitations of existing systems that focus only on detection of control-hijacking attacks, this project aims to develop a program transformation system called *DIRA* that can automatically transform an arbitrary application into a form that

- Can detect a control-hijacking attack when the control-sensitive data structure it tampers with is activated,
- Can identify the network packets that lead to the control-hijacking attack, and send these packets to a front-end content filter to prevent the same attack from compromising the application again, and
- Can repair itself by erasing all the side effects of the attack packets as if it never received them.

To the best of our knowledge, attack detection, repair and identification have never been considered together previously. Table 1 puts in perspective related projects in each of these three areas. The main contribution of this paper is the development of a unified solution to all three problems. Even though on the surface attack detection, identification, and recovery appear to be completely orthogonal functions, a careful examination reveals that they can actually be unified into a single implementation framework that is based on *memory update logging*. To repair a program's memory state, all updates to its address space should be logged so that these updates can be reversed. To detect a control-hijacking attack, the before image of a control-sensitive data structure should be stored away, and checked at the time of activation to see if any tampering took place. To trace back the packets responsible for a detected attack, the backward slice of the corrupted control-sensitive data structure needs to be first computed and then intersected with the incoming packets. *DIRA* takes an application's source code, and inserts additional logging code so that the resulting application can detect, identify, and recover from any control-hijacking attacks in a way that is completely independent of the underlying operating system and hardware.

The rest of this paper is organized as follows. Section 2 reviews previous research on detection and prevention methods for control-hijacking attacks, as well as on program rollback. Section 3 describes the logging algorithms and data structures used in the *DIRA* compiler. In Section 4 we discuss the implementation details of the *DIRA* compiler. Section 5 presents the performance measurements of a fully operational *DIRA* prototype and their analysis. Section 6 concludes this paper with a summary of major research contributions and a brief outline of the on-going work.

2. Related Work

Our work is based upon previous work in three broad areas of systems research: buffer overflow attacks detection, malicious code identification, and program rollback and replay.

Approaches to detect buffer overflow attacks can be divided into two groups: static techniques that detect potential buffer overruns by examining program's source code and dynamic techniques that protect programs at runtime. Wilander et. al. [39, 40] present a comprehensive overview of tools of both types. Greiner [16] gives an overview of manual code auditing techniques that help detect potential vulnerabilities.

The real cause of buffer overflows is unchecked pointer or array access. Jones and Kelly [20] and Austin et. al. [3] propose to check each pointer access at run time to solve this problem. This requires augmentation of the standard pointer representation with additional fields such as the extent of the memory region that the pointer is referring to. Both systems are implemented as C compiler extensions that instrument the source code of the program in such a way that the modified program checks each pointer access it performs at run-time. Purify [18] is a similar tool that instruments program's object code and therefore does not require access to its source code. However, all these tools suffer from a significant performance overhead which can be more than 500% in some cases. CRED [31] is a project that aims to provide a comprehensive memory access bounds checking at a reasonable cost. Unlike other bounds checking projects, CRED checks the access correctness for pointers to character strings only assuming that improper string manipulation is responsible for most buffer overflow attacks. The reported overhead of CRED is less than 26%. Such a moderately high overhead indi-

cates a need for more lightweight and inexpensive protection mechanisms.

The return address is the most common target of buffer overflow attacks. Stackguard [10] is a system that protects the return address by placing a *canary word* on the stack before the return address. It is based on the assumption that overwriting the return address requires overwriting the part of the stack immediately preceding it. If the canary word is found modified upon the function return then an attack has taken place. RAD [8] takes a different approach. It copies the return address to a buffer called the *return address repository* which is protected from both sides by applying `mprotect()` system call. Similarly, it compares the return address on the stack with the saved value and raises the red flag if the two values are different. StackShield [36], ProPolice [11], and StackGhost [14] are similar systems that protect other code pointers such as function pointers and stack frame register in addition to the return address. FormatGuard [9] provides a set of wrapper functions that protect a program from format string attacks.

Another approach to buffer overflow prevention is presented by Baratloo et. al. [4]. They develop a dynamic library called Libsafe that provides wrappers for common *libc* functions that are prone to buffer overflows. This library is transparently inserted at run-time between the application being protected and *libc* using `LD_PRELOAD` environment variable. The protection mechanism is based on estimating the boundaries of the stack frame of the calling function and assuming that no function can write below that boundary.

A typical buffer overflow attack executes the injected code on the stack. Therefore, making stack non-executable will prevent any stack-based attack. PaX [34] and Openwall [27] are two Linux kernel patches that implement non-executable stacks. This approach has some limitations, however. First, attacks that inject their code into data segment as well as *return-into-libc* attacks will still work. Second, Linux signal handlers and some functional languages such as LISP require the stack to be executable.

Essentially, the problem of detecting a buffer overflow attack relies on a mechanism to monitor a particular memory location (such as a return address). A similar problem exists in software debugging in which case a dynamically monitored memory location is called a watch-point. Existing solutions of this problem can be divided into runtime dynamic checking techniques [17, 18] and hardware-based techniques [25, 30, 41].

The problem of automatic identification of malicious code became increasingly important in the past few years since worms epidemics started to happen more and more frequently and at higher speeds. Given the speed of propagation of the recent worms, it is hopeless to rely on a human-based methods for signature generation as by the time the proper signature is created and distributed among computer users, the worm is likely to infect a significant number of computer systems. Autograph [21] is a system that generates worm signatures automatically by detecting common byte sequences in suspicious network flows. In this system, a network flow is considered suspicious if it comes from a host that is believed to perform port scanning. Toth and Kruegel [35] propose a system that detects malicious code in packet payloads by performing abstract execution of the payload data. Buttercup [29] is a system aimed at preventing polymorphic worms with known signatures from entering the system. It identifies the ranges of possible return addresses for existing vulnerabilities and checks whether a network packet contains such addresses.

Another approach to identifying malicious code is to analyze the execution trace of a compromised program. Given the address of the compromised control-sensitive data structure, one can use dynamic slicing techniques [24, 22, 23] to find out all statements of the program that affected the value of this data structure. This allows one to trace back the origin of the malicious data that was written to this data structure to the point where it first appeared in the program. Therefore, one can completely restore the compromising network packet or user input. Agrawal and Horgan [2] discuss several approaches for computing dynamic slices and introduce the notion of a dynamic dependence graph.

Finally, yet another approach to malicious input identification it to use a technique similar to Perl taint mode. The idea is to assign different tags to all user inputs and propagate these tags along through all memory operations. Upon discovering a compromised data structure, one can identify the origin of the malicious data by looking at the tag currently associated with that memory location.

System support for rollback and reverse execution is another related area of systems research. Although not related directly to post-attack recovery, these mechanisms can be readily adapted to rollback a program to a pre-attack state. Systems that have a rollback capability rely on one of the following techniques: they either keep the execution history [1] or do periodic state checkpointing [13, 28, 33]. For example, Igor [13] is a system that saves modified memory pages at each checkpoint. RECAP [28] and Flashback [33] use copy-on-write `fork()` system call to checkpoint their execution state. Spyder [1] is based on the notion of execution history. During its normal execution, Spyder records the program counter and the old values of all variables that the current instruction will change. All these systems require specific support from the underlying OS.

An alternative way of bringing a compromised system to the normal state is a complete restart. Candera et. al. [7, 6] develop the concept of micro-reboots. According to this concept, a complex system comprised of many individual components (such as a large Internet service) can be efficiently repaired in case of a fault or an attack by performing a micro reboot of a single failed component rather than that of the whole system. If the problem cannot be fixed by micro-rebooting then it is deferred to human operators.

3. Attack Detection, Identification, and Repair

DIRA makes programs capable of attack detection, identification and repair by using a combination of static and dynamic techniques. At compile time, the *DIRA* compiler instruments the source code of a program in a number of ways. First, it inserts proper memory updates logging code that allows the program to keep track of every memory update it performs. Second, the *DIRA* compiler inserts the code that checks every control-sensitive data structure before it is used. Finally, a number of special functions that allow the program to identify attack packets and repair itself are added to the program. At run time, the instrumented program generates a memory updates log which can be used to identify attack packets and repair the program once an attack is detected. The logged information is also used to check the control-sensitive data structures at run time when they are about to be used. If a control-sensitive data structure is found compromised, the attack identification and repair functions are called.

The amount of logging information as well as the type of information stored in the log depends on the mode in which *DIRA* operates. There are three modes of operation: compilation to support attack detection only (D-mode), compilation to support detection and identification (DI-mode) and finally compilation to support detection, identification, and repair (DIR-mode). Each successive mode requires more information to be logged. In this section we describe how memory update logging works and how the logged information is used in attack detection, identification, and repair.

3.1. Attack Detection

Most of the control-hijacking attacks modify some control-sensitive data structures in the victim program, such as a return address, a function pointer, or a jump table, through buffer overflowing. Once the compromised data structure is used in a control transfer, the attacker hijacks the control of the application.

The approach to attack detection used by *DIRA* is similar to that developed in RAD project [8]. To detect control-hijacking attacks at run time, the *DIRA* compiler maintains the original image of every control-sensitive data structure, and at the time of control transfer compares the current value of the associated control-sensitive data structure with its original image to determine whether it has been modified via buffer overflowing. Current version of the *DIRA* compiler protects only return addresses and function pointers as they are the most common attack targets. In particular, the *DIRA* compiler instruments an input program as follows:

- At the function prologue, the return address is stored in the return address buffer. At the function epilogue, the return address on the stack is compared with the stored value in the return address buffer. If there is a mismatch, the return address has been tampered with and a control-hijacking attack is detected.
- Every time a function pointer is modified in the program, its newest value is stored in an existing or new entry of the function pointer buffer. This includes the case when a function pointer is passed as an input argument into a function. There are also other ways to modify a function pointer, for example by overwriting it using `memcpy()` function. The current version of the *DIRA* compiler supports only simplest kind of function pointer modification when it is updated through a direct assignment. Each entry of the function pointer buffer contains two fields: the address of a function pointer variable and its value. Every time a function pointer is about to be used in a function call, its current value is checked against the function pointer's stored value. The mismatch of the two values is the indication of an attack taking place.

Because the return address buffer and the function pointer buffer are supposed to contain the ground truth, they should be well protected such that tampering via buffer overflowing is impossible. Otherwise, if an attacker can overflow both a control-sensitive data structure and its associated duplicate buffer, she can defeat this attack detection method. Towards this end, both the return address buffer and the function pointer buffer are sandwiched inside a pair of read-only pages. Any attempts to modify these two buffers via overflowing will result in protection faults. An attacker might try to compromise the system by guessing the address of such a buffer and writing to it directly without having to go through the protected pages. In order to reduce significantly the likelihood of a successful attack, such a buffer can be allocated at a random memory location. This technique is not implemented yet in the current version of *DIRA*.

In theory, the *DIRA* compiler can also protect jump tables in the same way as function pointers. However, because there have never been any real control-hijacking

attacks that tamper with jump tables, for simplicity we chose to ignore jump table protection in the current prototype.

3.2. Memory Updates Logging

The purpose of memory updates logging is to make it possible to trace back incoming packets that are responsible for the detected attack, and to restore the victim program back to the state before the attack packets were received. Keeping a copy of a control-sensitive data structure for attack detection can be seen as a special form of memory updates logging. To associate corrupted control-sensitive data structures with incoming packets, for each incoming packet it is necessary to compute all data variables that directly or indirectly depend on the incoming packet. If a corrupted control-sensitive data structure is data-dependent on an incoming packet, the packet is considered an attack packet and its contents will be used for content filtering.

To allow a program to be rolled back to the state it was in before receiving an attack packet, a snapshot of the program's state should be checkpointed every time it receives a packet. However, the performance overhead of this approach is too high to be feasible. Instead, *DIRA* uses a fine-grained asynchronous checkpointing approach, which logs updates only to global or static variables, and performs these logging operations in an incremental fashion rather than in one batch. When choosing this approach, *DIRA* assumes that during the interval between when a control-hijacking attack is detected and when the corresponding attack packet is received, the program will not be able to undo any file or network I/O operations. Therefore, memory updates logging can only erase attack's side effects on the memory state of the program, but not on its file system state.

Each record of the memory update log has four fields: `read_addr`, `write_addr`, `len`, and `data`. We will describe the meaning of each field below.

Most if not all control-hijacking attacks use one or multiple network packets to overrun a buffer in the victim program and eventually overwrite some control-sensitive data structures. Therefore, the content of the corrupted control-sensitive data structure is derived from the attack packets through a series of memory copying operations. Memory image of a program can be changed by either the program itself or by a library function call made by the program. To handle the updates of the former type, *DIRA* logs the effects of assignment statements of the following form: `X = Y`, where `X` and `Y` are directly referenced variables, array references (e.g., `a[i]`) or de-referenced variables (e.g., `*(a+1)`). The *read address* field contains the address of the right-hand-side variable of the assignment operation, in this case `Y`'s address. The *write address* field holds the address of the left-hand-side variable being modified, in this case `X`'s address. The *length* field is the size of the modified variable, size of `X` in this case. The *data* field is not used when *DIRA* operates in DI-mode. In DIR-mode, this field stores the pre-image of `X`, the variable being written to. It is not always possible to uniquely identify the read address, for instance if `Y` is a complex expression containing a number of variables or a function call. In this case the read address is set to "-1," which indicates that the data origin of this assignment is unknown.

The second source of memory image changes are standard library calls such as `memcpy()`. *DIRA* proxies several *libc* functions that can change program's memory state. Whenever a function that is proxied is called, the corresponding proxy function also produces a memory updates log record that summarizes the side effects of the function call. The fields of the memory updates log entry are set differently for different proxied functions. We will discuss all the functions proxied by *DIRA* below.

Some programs provide an alternative implementation of standard *libc* functions. Such functions will be compiled by *DIRA* thus providing necessary support for attack detection, identification and repair as long as these functions are written in standard C. If, however, the new *libc* functions are implemented using inline assembly (for performance reasons, for example), then *DIRA* will not be able to instrument them.

The above memory update logging algorithm implements both state checkpointing and data dependency tracking. Moreover, the *DIRA* compiler inserts logging code for each assignment operation of the form specified above without performing any sophisticated data or control flow analysis. As a result, the implementation complexity of the *DIRA* compiler is greatly simplified.

To reduce memory updates logging overhead, *DIRA* tries to avoid unnecessary logging operations. In its default mode, *DIRA* does not log updates of the form `X=Y` if `X` is a local variable referenced directly. The reason behind this is the assumption that local variables referenced directly are usually used as temp variables (for example, as loop variables) and do not contain any data coming from external sources. Although in general this approach can miss certain dependencies, it turns out that quite often enough information is logged to identify the malicious input.

Figure 1 shows the abbreviated source code of a simplified network service application containing a buffer overflow vulnerability. To ensure that omitting local updates logging is usually harmless, let us consider the logging operations performed by the program compiled in DIR-mode and determine that this information is sufficient to identify the attack packet. Function `do_packet()` calls function `get_packet()`, which receives a packet by call-

ing `recv()`. This is one of the functions proxied by *DIRA*. The corresponding proxy function logs the pre-image of `buf` by setting the `write_addr` field to the value of `buf`. Then the control flow goes back to function `do_packet()`. The assignment of the returned value to variable `last` is not logged because `last` is a local variable referenced directly. Function `process_packet()` is called next taking `last` as its only argument. The assignment of `buf` to `packet` is not logged either because `packet` is also a local variable referenced directly. Next there is a call to function `strcpy()` which is also proxied by *DIRA*. This function copies some data from `packet` into a limited-space buffer `name` without checking the length of `packet->name`, and thus represents an attack target. The `read_addr` field of the corresponding log record is set to the address of `packet->name`. To summarize, the memory update log contains two entries relevant to the packet being processed. None of the intermediate assignment operations are logged. Nevertheless, it is still possible to identify the packet that should be held responsible when an attack is detected. Indeed, variable `packet` of function `process_packet()` contains the same address as variable `buf` of function `get_packet()`. All intermediate assignments transfer the *value* of the pointer that points to the buffer containing the attack packet.

Memory updates log contains additional information when *DIRA* works in the DIR-mode. This mode requires storing additional information in the log such as marks that indicate function boundaries and potential restart points. We call such records *tags*. There are several types of tags. The tag type is stored in the field `read_addr`. The remaining fields are used differently for each tag. We will describe each tag type one by one in Section 3.4.

3.3. Attack Identification

Upon detecting a control-hijacking attack, we assume that the corrupted control-sensitive data structure is compromised by some data that might have been read from the console by a `gets()` call or from a network socket by a `recv()` call. In these cases, it is important to identify the source of corruption and take proper measures to prevent the same compromise from happening again. Of course, it is also possible that the control-sensitive data structure was actually overwritten due to a mistake in the program's internal logic. In this case, the program should be just terminated since no automatic repairing can stop the same compromise from recurring.

To identify the data item read from the network or a file that is responsible for the corruption of a control-sensitive data structure, we need to trace back the dependency graph, starting from the corrupted control-sensitive data structure. This tracing relies on the read address and write address fields of the memory updates log entries. Let *MA* (modified address) be the address of a corrupted control-sensitive data structure, i.e., a return address or a function pointer. In most cases, it was tampered with as a result of an unchecked array-to-array copy operation such as `strcpy()`. Each of such modifications leaves a record in the memory updates log. Therefore, the tracing begins with the most recent memory updates log entry whose write address is equal to *MA*, and uses the read address field of this entry as a key to search the memory updates log to find the most recent log entry whose write address matches it, etc. This process continues iteratively until reaching a memory updates log entry whose read address is set to one of the special values described below, which means that the data written to the write address of that entry comes from an external source. The above trace-back algorithm is formally described in Figure 2.

To support attack identification, the following classes of *libc* functions need to be "proxied": *copying/concatenation* functions such as `strcpy()`, *network I/O* functions such as `recv()`, *file I/O* functions such as `read()` (which can also read data from network), and format string functions such as `sprintf()`. The complete list of functions proxied by *DIRA* including those required for post-attack recovery only is presented in Table 2.

Copying/concatenation Functions. Each proxy function from this group generates a log record. For instance, a log record for a `strcpy(a, b)` function call contains the address of `b` in its `read_addr` field, the address of `a` in its `write_addr` field, `strlen(b)` in its `len` field. The `data` field is set to NULL as no data is required for the traceback algorithm. After generating a log record the proxy function calls the corresponding *libc* function and returns its result.

Network I/O and File I/O Functions. These proxy functions also generate one log record each time they are called. The `read_addr` field of these records is set to a special value indicating the external source of the data being logged. In addition, they make use of `data` field of the log record. This field stores the post-image of the memory buffer, that is, the data that was actually read from the network or a file. This data is the malicious network or file data that can lead to a buffer overflow attack. It is presented as the result of the traceback algorithm if a buffer overflow attack is detected. This data can be sent to a front-end intrusion-detection system, which can then use it to prevent the same attack from reaching internal hosts again. This automatic attack packets extraction capability protects an enterprise from worm-like attacks, where attacking or compromised hosts tend to send out attack packets that are largely the same.

```
void do_packet() {
  packet_t *last;
  last=get_packet();
  process_packet(last);
}
```

```
packet_t *get_packet() {
  char *buf=malloc(PACKSZ);
  recv(sock, buf, PACKSZ, 0);
  return buf;
}
```

```
void process_packet(char *buf) {
  char name[10];
  packet_t *packet;
  packet=(packet_t*)buf;
  strcpy(name, packet->name);
}
```

Figure 1. An example of a program vulnerable to a buffer overflow attack.

```
cur_addr=MA;
while (more_log_entries && cur_addr≠0)
   ent=get_prev_log_entry();
   if ent.write_addr ≤cur_addr && ent.write_addr+ent.len>cur_addr
      then cur_addr=ent.read_addr+(cur_addr-ent.write_addr);
end;
if (cur_addr≠0)
   { printf("Can't find source of attack\n"); exit(0); }
/* ent is the required log entry */
```

Figure 2. The traceback algorithm used to locate the source of a buffer overflow attack based on a corrupted control-sensitive data structure.

3.4. Attack Repair

Although *DIRA*'s attack detection mechanism can successfully prevent a control-hijacking attack from taking over a victim application, the application itself may need to be terminated as a result of such an attack. This "terminate and restart" approach to recover from a control-hijacking attack is not always desirable. Instead, it is better if the victim application can simply erase the effects of the attack packets as if these packets never happened. There are two issues involved in this program state repair process: (1) From which state should a victim program restart? (2) How to restart a victim program without special OS support?

Because *DIRA* logs only updates to global and array-like variables, it can only restart a program from the entry point of a function. The proper function `f_restart` turns out to be the *least common ancestor* of the function in which the attack was detected and the function in which the malicious external data was read in. We will call the function called from `f_restart`, which eventually led to malicious data read operation `f_read` and the function that eventually led to the attack `f_attack` (both can be the same function or even `f_restart`). The reason behind choosing `f_restart` the way we described it above is the fact that the stack frame of the dynamic parent of `f_restart` has not changed between the point when the malicious data was read in and the point when an attack was detected whereas the stack frame of any other function called after `f_restart` as well as `f_restart` itself might have changed between the two moments. Since we do not track any local variable updates we will not be able to bring the program back to a consistent state older than the state in which it was right before `f_restart` was called. There is an exception from this rule, however. If there are no local variable updates in `f_restart` between the point after `f_read` returns and before `f_attack` begins, we can safely restart the execution from `f_read` instead of `f_restart`.

Sometimes, it is still possible that the whole program will need to be restarted. Indeed, this happens if `f_restart` turns out to be function `main()` and there are some local variable updates made between when `f_read` returns and `f_attack` begins. While evaluating *DIRA* we have encountered one such program. One way to avoid this problem is to track all variable updates including local ones, but that may significantly increase the run-time overhead. If the repair algorithm finds that the program needs to be restarted from the beginning, the program is simply terminated and restarted afterwards.

Identifying the function that reads in the malicious external data is a part of attack identification process and therefore does not incur any extra run-time overhead while the program is running normally. Figure 3 illustrates how the restart point is chosen with a typical buffer overflow attack scenario. In this case, either `f2()` or `f1()` can be chosen as the new restart point. The decision depends on whether there are any local variable updates in `f1()` after return from `f2()` until the call to `f4()`.

DIRA does not require any system support for program restart. Instead, it uses inter-procedural jump functions

setjmp() and longjmp() to implement this functionality.

Figure 4 shows the algorithm that *DIRA* uses to find the least common dynamic ancestor between the function that detects an attack or a corruption of some control-sensitive data structure, and the function that inputs the malicious data from the memory updates log. Logically, the algorithm traverses the memory updates log backwards to find the first function whose function entry tag is earlier than the function entry tag of both functions. The algorithm includes two steps. The purpose of the first step is to find out the depth of f_read with respect to the least common dynamic ancestor of f_read and f_attack. In order to determine this depth the memory updates log is traversed in backwards direction. The traversal starts from the last log entry and continues until the function entry tag of f_read is reached. Variable depth is a loop invariant and has the following meaning. It contains the relative depth of the function which the current log record belongs to with respect to the greatest dynamic ancestor function of f_attack seen so far. It is obvious that the traversal will sooner or later go through a log entry that belongs to the least common dynamic ancestor of f_read and f_attack because the control flow should have returned to this function at least once between the point when f_read was called and the point when f_attack was called. Therefore, depth will eventually contain the relative depth of f_read with respect to the least common dynamic ancestor. At each loop iteration, variable depth is updated as follows. If the tag of the current log record is a function entry tag and the depth equals 0 then the next log record to be traversed corresponds to a dynamic parent of the current function, and therefore it becomes the current greatest dynamic ancestor in which case the value of depth does not need to be changed. If, however, the tag of the current log entry is a function exit tag then the function which that log entry belongs to was called from the current function. Therefore, we need to increase depth by one. Finally, if the tag of the current log entry is a function entry tag and the depth is greater than zero then we need to decrement depth by one as this means that the current function is a dynamic child of the greatest ancestor function. To summarize, at the end of the first step variable depth equals the relative depth of f_read with respect to the least common dynamic ancestor. All we need to do after that is to traverse the log backwards until we reach a function whose relative depth with respect to the least common ancestor is zero. The beginning of this function is the beginning of the least common dynamic ancestor of f_read and f_attack.

Finding a restart point requires augmentation of the memory updates log with several types of tags which are inserted to the log when the program runs normally. These tags are *function entry* tag, *function exit* tag, *jump buffer* tag, and *first local update* tag. Upon entering a function *DIRA* inserts a function entry tag into the memory updates log. Similarly, when the function returns a function exit tag is inserted. When a function call is made, *DIRA* inserts a call to setjmp(buf) where buf is the data field of a memory log record. The read_addr of this record is set to the jump buffer tag. This makes the point preceding the function call a potential restart point. At repair time, the control can be transfered to this point by performing longjmp(buf). Finally, the first local update tag is inserted to the log when the first update to a local variable is encountered after a function call. These tags are used at repair time to determine the actual restart point, which can be either f_read if no such tags are found in f_restart between the call to f_read and the call to f_attack, or f_restart if at least one local update tag was found.

Once the restart point is determined, the memory state of the program needs to be rolled back to the state corresponding to the new execution point. To do so, the attack recovery module needs to traverse the memory updates log in the reverse direction until it reaches the restart point, undoing each global variable update along the way. After the undo, the recovery module performs a longjmp() using the jmp_buf corresponding to the restart point.

To restore the memory image of the program the complete pre-image of each memory update should be stored in the data field of the corresponding log record. The same is also true about *libc* functions proxied by *DIRA*. For instance, strcpy(a, b) call needs to store the pre-image of buffer a of length strlen(b) in the data field of memory updates log record. In addition, several other classes of *libc* functions need to be proxied. They are: memory management functions, privilege management functions, process management functions, and inter-procedural jumps functions. The complete list of proxied functions is presented in Table 2. Below we will consider each group of functions in more detail.

Memory Management Functions. Each function in this group is proxied for the following reason: at repair time, the program needs to be able to undo not only global memory changes, but also memory manipulation functions it called before. The proxy_malloc() calls malloc() first and stores the address of the newly allocated object in the memory updates log. At repair time if this record needs to be rolled back, this memory object is freed.

During repair time we also need to reallocate objects that were previously deallocated. This is achieved by proxying free(). A straightforward way to restore the object that was deallocated is to allocate it again with malloc(). However, the new object may be created at a new memory locations and all earlier references to it in

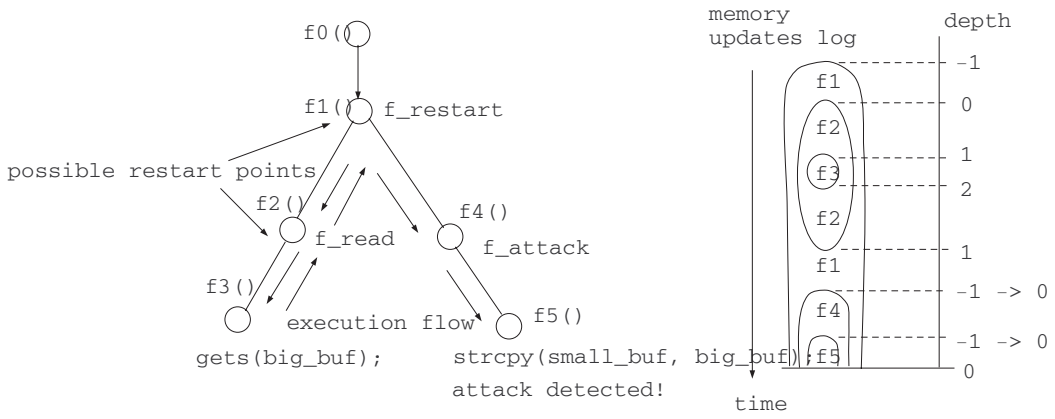

Figure 3. An example illustrating how to identify the least common dynamic ancestor in the function call graph and use it as the restart point. The right-hand side shows the memory updates log where f1–f5 are the same functions as those on the left-hand side. The ovals correspond to function boundaries. The depth values are the values of depth variable that is defined in Figure 4.

```
f_read — function in which malicious data was read in;
ent_beg=first log entry of f_read (function entry tag);
depth=0;
ent=last_log_entry();
while (ent!=ent_beg)
   if (ent.tag is function entry tag) then depth−−;
   if (ent.tag is function exit tag) then depth++;
   if (depth<0) then depth=0;
   ent=get_prev_log_entry();
end;
/* second phase */
while (depth≥0)
   if (ent.tag is function entry tag) then depth−−;
   if (ent.tag is function exit tag) then depth++;
   ent=get_prev_log_entry();
end;
```

Figure 4. Algorithm for finding the least common dynamic ancestor in the function call graph.

Function class	Libc functions
Copying/concatenation	memcpy(), mempcpy(), memmove(), strcpy(), strncpy(), strcat(), strncat(), bcopy()
Network I/O	readv(), recv(), recvfrom()
Inter-procedural jumps	setjmp(), longjmp()
Memory management	malloc(), calloc(), realloc(), free(), strdup()
Privilege management	seteuid(), setreuid(), setegid(), setregid()
Process creation	fork()
File I/O	read(), fread(), scanf(), vscanf(), fscanf(), vfscanf(), gets(), fgets()
Format string	sprintf(), snprintf(), vsprintf(), vsnprintf(),

Table 2. The set of functions that *DIRA* needs to proxy to support attack identification and repair.

the memory updates log will need to be remapped. Instead, we use a *deferred free()* approach. When the program calls `free()`, the `proxy_free()` function just puts the address of the object into the log without freeing up the object. At repair time, we do not need to do anything to restore the original object since it is kept in the memory.

Finally, the `proxy_realloc()` function saves the original pointer in the buffer, replaces the original `realloc()` call with a `malloc()` call and saves the pointer to the newly allocated memory as well. Then it copies the data to the newly allocated buffer. The length of the data being copied is obtained from the memory buffer header that is preceding the data itself. At repair time, the newly allocated object is deallocated.

The description above suggests that during its normal execution the program will never free up the memory it allocates. However, since the capacity of the memory updates log is limited, its records are used in a circular fashion. A single record can be reused if the program runs long enough. When a record is reused, its previous content is cleaned up and the recovery of that operation becomes impossible. A part of the clean up procedure for a `proxy_free()` buffer entry is the `free()` operation that was deferred before.

Inter-procedural Jump Functions. Function `longjmp()` performs an inter-procedural jump to one of the dynamic ancestors of the current function. To keep the memory updates log consistent, we need to add a proper number of function exit tags to the log. This number equals the number of functions skipped by `longjmp()`. To determine it at run time, we proxy both `setjmp(jmp_buf)` and `longjmp(jmp_buf, state)`. The `proxy_setjmp()` function logs the address of the `jmp_buf` variable. The `proxy_longjmp()` function searches the memory updates log for a log entry corresponding to `setjmp()` call that filled in the `jmp_buf` used in `proxy_longjmp()`. Once it finds the proper log entry, it can find out the nesting level of the current function with respect to the target function and thus compute the required number of function exit tags to be added.

Privilege Management Functions. Many programs change their effective user ID and group ID values for security reasons. At repair time, the proper values need to be restored to give the program same access rights as those it had at the chosen restart point. This is achieved by proxying functions such as `seteuid()` and `setegid()`. These functions save the original value of uid or gid in the `data` field of a memory updates log record. The `read_addr` field holds the proper proxy function tag that allows the repair procedure to identify such log records and call appropriate privilege management functions with the values stored in the `data` field. A privileged process calling `setuid()` can replace its effective user ID with a nonzero effective user ID in which case the old effective user ID can never be restored. In this case, there is no way for the recovery process to restore the original effective user ID without explicit system support.

Process Management Functions. When a program compiled by *DIRA* forks a new process, the two processes can access their memory updates logs concurrently because of the copy-on-write semantics of `fork()` system call. In this case, two versions of the log are created automatically by the OS. However, if a buffer overflow attack is detected in one of the processes then the repair procedure might require the program to be rolled back to the point before a new process was forked. The current version of *DIRA* does not consider the problem of cascading rollback. Instead, if the overflow was detected in the parent process then all child processes that have been forked after the new restart point are killed. However, if an attack is detected in the child process and the restart point is chosen to be before the point where it was forked, the process is terminated without affecting the parent process in any way. The `proxy_fork()` function inserts special tags in both parent's and child's process logs to facilitate this process.

3.5. Limitations

The memory updates logging algorithm currently used in the *DIRA* compiler is designed for simplicity, and thus has much room for performance optimization. For example, because the current *DIRA* compiler only tracks data dependencies carried by simple assignment operations and proxied functions, it cannot identify dependencies that involve any arithmetic expressions, e.g., `B=A+C`. This means that *DIRA*'s recovery module may not be able to trace a corrupted control-sensitive data structure back to a malicious network packet if the former is derived from the latter through any form of transformation other than assignment operations. It is possible to generalize the current memory updates logging algorithm by leveraging information from data flow analysis techniques that allow *DIRA* to identify and log those and exactly those statements that may be data dependent on the network packets.

Data/control flow analysis can also improve the efficiency of state checkpointing. Ideally, the pre-image of each global or static variable needs to be recorded exactly once for each logical checkpoint. However, because a function may be called from different places and the current *DIRA* compiler does not perform inter-procedural control flow analysis, the pre-image of each global or static variable used in a function is recorded at least once per invocation of that function. Furthermore, due to aliasing, the pre-image of the same global or static variable may be logged multiple times within a function invoca-

tion. Data/control flow knowledge can help eliminate unnecessary recording of pre-images, e.g., when a global variable is repeatedly updated within a loop.

The current version of *DIRA* can only handle concurrent accesses to the memory updates log from processes that are launched through `fork()` system call. However, it does not recognize other forms of fork such as `vfork()`. In these cases, some form of locking mechanism is required to provide exclusive access to the memory updates log. It is also possible that a program performs some file or network I/O operations between the point when the attack packets were read in and when the attack was detected. In this case, restoring the memory image of the program without restoring the underlying file system state might lead to an inconsistent state of the program. Ideally, both memory and file system repair should be performed. The current *DIRA* prototype does not support file system repair.

Signals are frequently used in network daemons as a means of scheduling exceptional events. Currently, *DIRA*'s repair mechanism does not support undo of signals. Ideally, all signals that were set after the new restart point need to be canceled.

4. Implementation Issues

4.1. Source Code Instrumentation

The current *DIRA* prototype is implemented as an extension to GCC 3.3.3. When compiling a program, GCC converts the source code to a number of representations. First, it translates the program into an Abstract Syntax Tree (AST). Then, the AST representation is converted into a Register Transfer Language (RTL) representation. Finally, the RTL code is converted into the machine code for the target platform. *DIRA* instruments the source code at two levels: AST level and machine language level. The latter is used to generate special prologue/epilogue code that supports return address defense as well as inserts function entry/exit tags. Currently, *DIRA* supports only the IA-32 platform.

The code required to support memory updates logging is inserted to the source code directly when it is represented as an AST. *DIRA* converts each tree of type `MODIFY_EXPR` representing an assignment operation `X=Y` into a compound tree of type `COMPOUND_EXPR` that is equivalent to the following C code: `(log(X,Y,sizeof(Y)), X=Y);`. Such a transformation can be inserted at any place in the source code.

All unary arithmetic operations (such as `++` and `--`) contained in the original expression are stripped off when the logging call is made to avoid repeated variable modification.

To proxy necessary function calls *DIRA* checks all `CALL_EXPR` trees in the original program. If the function name of the function call is one of those that need to be proxied, *DIRA* replaces the original function name with the proxied function name. In order to support restart points, *DIRA* replaces the original `CALL_EXPR` with a `COMPOUND_EXPR` that first makes a call to `setjmp()` to insert a restart point and then makes the original function call.

4.2. Transparent Library Compilation

Often programs need to be linked with several non-standard libraries. At the same time, same libraries can be reused by multiple applications. Some applications (typically server-side applications) might need *DIRA* support whereas others (typically client-side applications) will most likely not. Therefore, both instrumented and uninstrumented versions of some libraries should exist in the system. A naive approach would be to have two versions of the same library under different names. However, in this case one will have to go through all the Makefiles of the program and change every occurrence of the name of the old library to that of the instrumented library if the program needs *DIRA* support. Another drawback of this approach is the fact that a program would not be able to switch from one version of the library to another without being recompiled.

A better solution of this problem is to duplicate the code of every function in all the source files that constitute the library. The first copy of the function is instrumented whereas the second one is left intact. *DIRA* inserts an if-statement in the beginning of each function and makes the two copies of the original function its `then` and `else` branches. The if-statement checks whether the following condition is true or not: `need_logging==NULL` or `*need_logging==0`, where `int *need_logging` is a special variable inserted by *DIRA* to the original program. If the condition is true, then the unmodified version of the code is executed. Otherwise, the control flow is transfered to the instrumented version. When an application that does not require *DIRA* support is linked with a *DIRA*-compiled library, the control will always be transfered to the uninstrumented version of the code since the application is not aware of the `need_logging` variable which is set to `NULL` by default.

This code duplicating approach introduces several implementation issues. The first issue is related to code generation for switch statements. GCC creates a set of labels (trees of type `CASE_LABEL`) for each case expression of the switch statement. When the second copy of the function is compiled, these labels are reused instead of being recreated. As a result, the control is transfered to the first version of the code that was compiled before. The solution to this problem is to recreate the labels each time a

switch statement is processed regardless of whether they were created before or not. The second issue is related to a common optimization performed by GCC. When it generates RTL code for a complex AST expression, GCC wraps this AST into a tree of type `SAVE_EXPR`, which indicates that there is an RTL code available for the original AST expression and there is no need to process it again. However, *DIRA* requires that all ASTs be processed twice since otherwise the logging code might not be generated for the second copy of the AST. To ensure that this is the case, *DIRA* wraps the trees of type `SAVE_EXPR` into the trees of type `UNSAVE_EXPR` and nullifies the effect of `SAVE_EXPR`.

Sometimes, a program needs to be linked with a library which source code is not available. In this case, *DIRA* will still be able to compile and link the program, but all memory manipulations that happen inside the uninstrumented library will not be tracked. As a result, it might not be possible to detect an attack if one of the uninstrumented functions is responsible for it or to identify it since the variable dependency chain might be broken because of a call to an uninstrumented function. Also, it will be impossible to undo the side effects of such functions at repair time which can lead to memory leaks or even worse to an inconsistent memory state of the program.

5. Evaluation of DIRA

5.1. Performance Overhead

In this section we present the evaluation of *DIRA*. We are interested in its compilation time overhead, executable code size increase, and several run-time characteristics such as the amount of log information generated and the performance overhead when a program is compiled in three different modes of *DIRA*: D-mode, DI-mode, and DIR-mode. We also describe our experiences with attempts to compromise programs compiled by *DIRA* and discuss whether repairing the programs is worthwhile at all or restarting them from the beginning is a better strategy.

We used a test suite of five network daemons in our experiments: `ghttpd` 1.4 — an http server, `drcatd` 0.5.0 — a remote cat daemon, `named` 8.1 — DNS daemon which is a part of BIND program, `qpopper` 4.0.4 — a POP3 server, and `proftpd` 1.2.9 — an FTP server. We used several exploit programs for the three programs of our test suite: `named`, `ghttpd`, and `drcatd` available at Fyodor's Remote Exploit Archive [15] and Securiteam's website [5]. Our experiments proved that the instrumented versions of these programs can detect attacks, identify attack packets and continue their execution normally after repair (the latest is true for 2 out of 3 programs that we tried to compromise).

Program	Log records	Log size, KB
ghttpd	457	32
drcatd	4,000	408
named	832	39
qpopper	27,000	586
proftpd	70,000	2073

Table 4. Number of log records generated for a single client request as described in Table 3 as well as the total log size (KB) generated per client request.

The hardware setup used in the performance experiments is as follows. The network daemon being tested was running on a server machine with a Pentium-4M 1.7GHz processor and 512 MB of RAM. There were two client machines with AMD Athlon 1.7GHz processors equipped with 512 MB of RAM each. All machines were running the Linux 2.4.19 kernel. The machines were located in the same 100 Mbps local network. All programs were compiled on the server machine with options -g -O.

To measure several run-time characteristics of the instrumented server programs, the client machines were running special programs that were simultaneously sending a number of requests to the server machine. The description of the performance tests is given in Table 3.

The left table in Figure 5 presents the measurements of compilation time overhead, and shows that the compile time overhead varies significantly from program to program and can be between 130% and 550%.

The right table in Figure 5 shows the difference between the executable file size produced by *DIRA* and that by the original GCC compiler. Since *DIRA* duplicates the code of each function, one might expect that the executable produced by *DIRA* will be twice as large as the executable produced by standard GCC. This turns out to be the case for small programs such as `ghttpd` and `drcatd`, but does not hold for larger program. Most likely, this happens because a binary usually contains a number of sections such the data section, PLT section, symbol table section, etc. and the code section is just one of them. Also, GCC performs several optimizations to reduce the output binary size. This explains why the increase in binary size for larger programs is only 20-40% instead of expected 100%.

We conducted several series of experiments to measure different run-time characteristics of instrumented programs. First, we measured the number of memory updates log records as well as the total size of memory updates log (in KB) for a single client request as described in Table 3. The results are presented in Table 4. As the results suggest, the dependency between the number of log records

Program	Type	Client request	Repeated, times
ghttpd	HTTP server	fetch a 10KB HTML page	5,000
drcatd	remote cat daemon	fetch a 10KB file	1,000
named	DNS server	lookup of a domain name	10,000
qpopper	POP3 server	fetch a 1KB message	200
proftpd	FTP server	fetch a 40KB file	100

Table 3. Test programs and corresponding performance tests.

Program	GCC	DIRA	Overhead, %
ghttpd	0.82	3.77	359
drcatd	1.30	4.50	246
named	33.38	79.72	138
qpopper	11.58	26.73	130
proftpd	25.88	169.88	555

Program	GCC	DIRA	Overhead, %
ghttpd	87145	174778	100
drcatd	70126	156229	123
named	1452030	2036324	40
qpopper	1371275	1654643	21
proftpd	2257744	3113267	38

Figure 5. Increase in compilation time, sec (left) and the executable file size, bytes (right).

and the actual amount of data written to the log is not linear. The reason behind this is the fact that different log records have different actual size. The typical log records types are as follows. The size of a single variable update log record is 16 bytes (4 bytes for read address, 4 bytes for write address, 4 bytes for data length and 4 bytes the actual payload). A potential restart point log record has the size of 160 bytes because it contains a `jmp_buf` buffer used by `setjmp()` and `longjmp()`. Log records generated by proxied string manipulation and network *libc* functions can have different sizes. Typically, their length varies from several bytes to 1 KB.

The goal of our second series of experiments was to measure the run-time performance overhead of the instrumented programs which is certainly the most important performance metric of *DIRA* compiler. We compiled all five programs in three compilation modes that *DIRA* provides: D-mode that supports attack detection only, DI-mode that supports attack detection and identification, and DIR-mode that supports program repair in addition to attack detection and identification. The measurements from these experiments are presented in Figure 6 and suggest that the run-time overhead can vary significantly depending on the programs' memory access behavior and can range from 8% to 60% for programs that support attack detection, identification, and recovery.

The experiments showed that the run-time overhead of programs compiled in D-mode varies from 0% to 15%. We believe that this overhead is mostly affected by the frequency of function calls because it mainly comes from additional code in the function prologue and epilogue. That is, if functions are relatively long and called infrequently, then there is not much to do for the return address defense mechanism and the overhead can be close to zero percent. If, however, the program contains lots of small functions that call each other then the overhead can be much higher.

The run-time overhead of the programs compiled in DI-mode is much higher than that of the programs compiled in D-mode and is between 7% and 50%. Such a difference can be explained by the fact that a program needs to log certain information about its memory state changes such as the read address, the write address, and the length of the data being written. Programs compiled in DI-mode make use of proxy functions to track changes performed by standard *libc* calls. However, the pre-image of the data being modified is not saved in DI-mode and this is its only difference from DIR-mode.

The measured overhead of programs running in DIR-mode turns out to be pretty close to that of programs running in DI-mode due to the reason described above — there are relatively few differences between the two modes. In addition to saving the pre-image of data being modified, programs in DIR mode insert potential restart points by making `setjmp()` calls and also insert first local update tags.

The main conclusion from these experiments is that the run-time overhead depends on the programming style in which the program was written. There are certain things that can increase the run-time overhead such as breaking up the program into a large number of small functions and using pointer arithmetic extensively. These results also suggest a strong need for a more intelligent checkpointing mechanism that can help reduce the overhead. Instead of saving the data pre-image upon each update, one can save the pre-image of the whole data structure once upon function start.

We have also measured the relative frequency of each type of log records written to the memory updates log during a performance benchmark run. The results are presented in Figure 6. Although there seems to be no direct

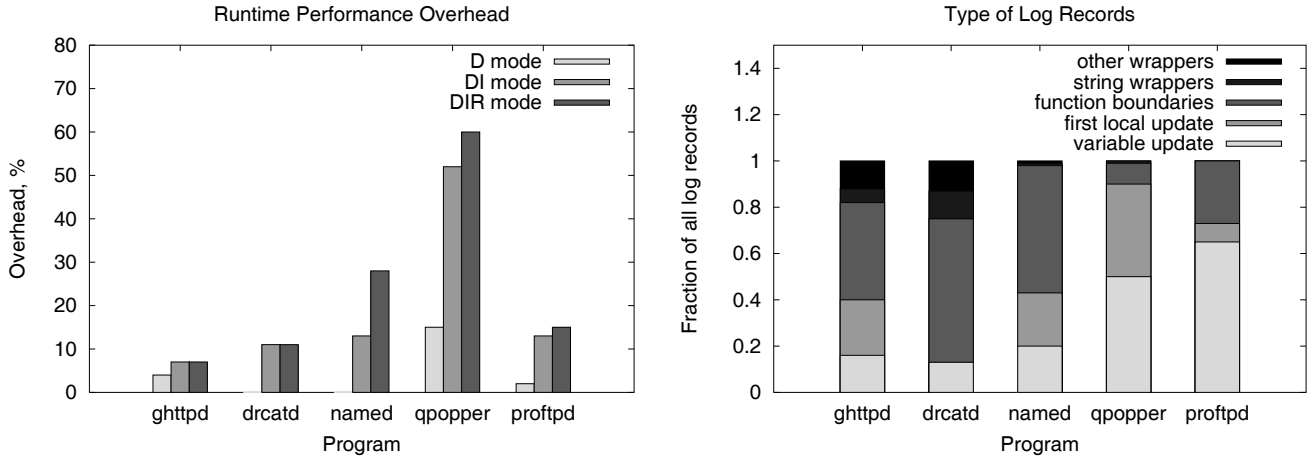

Figure 6. Run-time overheads for different modes of compilation (left) and the relative frequency of different types of log records (right).

correlation between the frequency of types of log records and the run-time overhead, these results still suggest that programs with higher overhead (such as `qpopper`) insert more records for variable updates and first local update tags. Indeed, these two types of records account for 90% of all records that `qpopper`) had written to the log. These results suggest once again a need for a more sophisticated checkpointing algorithm that can help reduce the number of variable update log records as well as a more intelligent mechanism for choosing the restart points. The current mechanism relies on information about local updates to determine where restart points are, and consequently requires generating a lot more first local update records than necessary. Ideally, such a mechanism should use dependency analysis techniques such as slicing to find out points in the program that can eventually lead to a function that reads external data. Only those points can be program's potential restart points.

In our next series of experiments we measured the amount of file and network I/O activity performed by the programs from our test suite. This information can help answer the question of whether the file system and network undo is indeed required for the repair process or the programs can be repaired and continue their execution without file system and network undo. The results are presented in Table 5. The results showed that 3 out of 5 programs that we tested do not perform any file output operations when serving a single client request. Our analysis of the source code of the remaining two programs showed that the file output operations performed by those programs are used to create temp files and write logging information. We believe that this information is not a critical part of program's state and therefore leaving it after an attack will not bring the program into an inconsistent state. The network output operations performed by the programs are related to communicating with the client that initiated the connection only. Therefore, if that client turns out to be malicious there is no need to undo the effects of network operations for such a client. These observations allow us to conclude that file system and network undo support is not really required for the network daemons that we have studied.

Program	File IN	File OUT	Net IN	Net OUT
ghttpd	45	0	1	49
drcatd	319	0	3	320
named	0	0	1	1
qpopper	41	80	5	7
proftpd	13	63	11	61

Table 5. Network and file I/O activity for a single client request as described in Table 3.

5.2. Experiences

We tried to compromise `named`, `ghttpd`, and `drcatd` compiled in DIR-mode by using malicious code from public databases [15, 5]. All these attempts failed because the programs were able to detect and identify the attacks. Moreover, two out of three programs could repair themselves to the extent that allowed them to continue normal execution. Below we describe our experiences of applying *DIRA* to each of these programs in more detail.

BIND named inverse query vulnerability. BIND named version 8.1 has a bug in its inverse DNS query pro-

cessing function allowing the attacker to gain root control of the vulnerable system. Malicious code available to us exploited this vulnerability by sending a single packet and waiting for response. It did not try to run a remote shell on the victim machine. For this particular program, the repair procedure determined that function `f_restart` is function `main()` of `named`. However, it turned out that there were no local variable updates from `f_read` to `f_attack`, and therefore the execution could be restarted from `f_read` (which was function `evGetNext()` in this case).

Format string vulnerability in ghttpd. There is a format string vulnerability in function `Log()`. The repair procedure determined that for this particular program functions `f_restart` and `f_read` is the same function `serverconnection()`. However, since there were a number of local variable updates between the point where the data was read and the point where `Log()` was called, the execution restarted from the beginning of function `f_restart`. Still, the initial connection was kept open. The exploit program that we used continued sending packets to the same port as where the initial malicious packet was sent to. These subsequent packets were treated as invalid requests by the survived program. It sent the "bad request" HTML page back to the exploit program in response.

Format string vulnerability in drcatd. There is a similar vulnerability in the logging function of `drcatd`. However, in this case function `main()` turned out to be `f_restart`. In addition, there were a number of local variable updates between `f_read` and `f_attack`. Therefore, the whole program needed to be restarted in case of an attack. This problem can be solved in two ways. The first is to reorganize the source code manually by putting potentially vulnerable parts of the code into a separate function so that the execution can be restarted from it in case of attack. However, this solution requires some understanding of the source code of the program and therefore is not suitable for automatic program protection. The second solution is to log all memory updates including local ones. However, current version of *DIRA* cannot tell automatically whether tracking global updates only will be sufficient or not. Currently, this option can be turned on and off manually. When compiled with this option turned on `drcatd` can repair itself and continue normal execution. However, even if it turns out that the whole program needs to be restarted the program can still detect the attack and identify it.

The analysis presented above shows that it is not always possible to repair a program automatically and that even when it is, the restart point may be quite close to the beginning of the program. This raises the question as to whether automated attack repair is useful in practice. We believe automated attack repair is indeed useful for the following reasons. First, with attack repair, dynamic attack detection is now as effective as static analysis in protecting vulnerable applications at run time without suffering from the latter's false positive problems. More concretely, even if a vulnerable application is compromised, *DIRA* ensures the application can continue as if the vulnerability does not exist in the first place. Second, automated attack repair is essential to provide protection among clients of single-threaded or event-based network applications. In these applications, requests from multiple clients are processed in the same process. Therefore, terminating an application of this type upon detecting an attack from some client is not acceptable as it also disrupts the service to other clients as well. Finally, *DIRA*'s automated attack repair can be used together with other types of attack detection methods such as system call argument monitoring, which can detect attacks that could damage a victim application's address space without hijacking its control. For these types of attacks, the automated attack repair mechanism can still repair the damaged address space even long after the attack takes place.

6. Conclusion

In this paper we presented the first known compiler that can transform arbitrary programs to a form that can detect control hijacking attacks, identify the malicious input and repair the compromised program, all without human intervention. In addition, the performance overhead of these transformations is shown to be quite modest, even without any aggressive optimizations.

There are a number of ways in which the *DIRA* prototype can be improved. First, we aim to improve the efficiency of the memory updates logging mechanism by employing control flow analysis. Currently, *DIRA* tracks every update to any global variable, even though in theory only the first one needs to be logged. Another problem with the current logging mechanism is that it may miss certain data dependencies, for example, when a local variable is used to transfer information between two global variables. Comprehensive data dependency analysis is required to improve the accuracy of attack identification.

We are going to address multi-threading issues in more detail in the next version of *DIRA*. Multiple threads of the same program can concurrently access the memory updates log and other global data structures, and thus introduce additional data dependencies. At repair time, *DIRA* needs to determine which threads should be rolled back, restore the state of each such thread to the corresponding pre-attack state, and resume its execution. We are also planning to broaden the scope of the repair process by including support for file system undo. Although not common among network applications, file system

undo can help erase the side effects of an attack on a file system such as temp files.

7. Acknowledgement

We would like to thank the anonymous reviewers and our shepherd Dr. Dawn Song for their valuable comments.

This research is supported by NSF awards ACI-0234281, CCF-0342556, SCI-0401777, CNS-0410694 and CNS-0435373 as well as fundings from Computer Associates Inc., New York State Center of Advanced Technology in Sensors, National Institute of Standards and Technologies, Siemens, and Rether Networks Inc.

References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. An execution-backtracking approach to debugging. In *IEEE Software*, May 1981.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1990.

[3] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. *ACM SIGPLAN Notices*, 29(6), 1994.

[4] A. Baratloo, T. Tsai, and N. Singh. Transparent run-time defense against stack smashing attacks. In *Proceedings of USENIX Annual Technical Conference*, June 2000.

[5] Beyond Security's SecuriTeam. `http://www.securiteam.com`.

[6] G. Candea and A. Fox. Crash-only software. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*, 2003.

[7] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. A microrebootable system — design, implementation, and evaluation. In *Proceedings of Operating System Design and Implementation Conference*, 2004.

[8] T.-C. Chiueh and F.-H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proc. of 21st Intl. Conf. on Distributed Computing Systems*, 2001.

[9] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. FormatGuard: Automatic protection from printf format string vulnerabilities. In *Proceedings of 10th USENIX Security Symposium*, August 2001.

[10] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, January 1998.

[11] H. Etoh. GCC extensions for protecting applications from stack-smashing attacks. `http://www.trl.ibm.com/projects/security/ssp`, June 2000.

[12] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the ACM SIGOFT Symposium on the Foundations of Software Engineering*, December 1994.

[13] S. Feldman and C. Brown. Igor: A system for program debugging via reversible execution. In *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, January 1989.

[14] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *Proceedings of the 10th USENIX Security Sumposium*, August 2001.

[15] Fyodor. Remote exploits. `http://www.insecure.org/sploits_remote.html`.

[16] L. A. Grenier. Practical code auditing. `http://www.daemonkitty.net/lurene`, 2002.

[17] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of Int. Conf. Software Engineering*, May 2002.

[18] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, 1992.

[19] S. C. Johnson. Lint, a C program checker. In *AT&T Bell Laboratories:Murray Hill, NJ*, July 1978.

[20] R. Jones and P. Kelly. Bounds checking for C. `http://www-ala.doc.ic.ac.uk/~phjk/BoundsChecking.html`, July 1995.

[21] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of USENIX Security Symposium*, 2004.

[22] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3), 1988.

[23] B. Korel and J. Laski. Dynamic slicing of computer programs. *The Journal of Systems and Software*, 13(3), 1990.

[24] A. D. Lucia. Program slicing: Methods and applications. In *Proceedings of the 1st IEEE International Workshop on Source Code Analysis and Manipulation*, 2001.

[25] S. L. Min and J.-D. Choi. An efficient cache-based access anomality detection scheme. In *Proceedings of the Fourth International Conferenceon Architectural Support for Programming Languages and Operating Systems*, 1991.

[26] J. Nazario. Project Pedantic — source code analysis tool(s). `http://pedantic.sourceforge.net`, March 2002.

[27] Openwall project. `http://www.openwall.com`.

[28] D. Z. Pan and M. A. Linton. Supporting reverse execution of parallel programs. In *Proceedings of the ACM SIGPLAN/SIGOPS Workhop on Paralleland Distributed Debugging*, January 1989.

[29] A. Pasupulati, J. Coit, K. Levitt, S. F. Wu, S. H. Li, J. C. Kuo, and K. P. Fan. Buttercup: On network-based detection of polymorphic buffer-overflow vulnerabilities. In *Proceedings of Network Operations and Management Symposium*, 2004.

[30] M. Prvulovic and J. Torrellas. ReEnact: Using thread-level speculation to debug software; An application to data races in multithreaded codes. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

[31] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Network and Distributed System Security Symposium*, February 2004.

[32] Secure software solutions. Rough auditing tool for security, RATS 2.1. `http://www.securesw.com/rats`.

[33] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of USENIX Annual Technical Conference*, June 2000.

[34] P. Team. Non-executable pages design and implementation. `http://pax.grsecurity.net/docs/noexec.txt`.

[35] T. Toth and C. Kruegel. Accurate buffer overflow detection via abstract payload execution. In *Proc. of Intl. Symposium on Recent Advances in Intrusion Detection*, 2002.

[36] Vendicator. StackShield GCC compiler patch. `http://www.angelfire.com/sk/stackshield`, January 2001.

[37] J. Viega, J. T. Bloch, T. Kohno, and G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *Proceedings of the 16th Annual ComputerSecurity Applications Conference*, December 2000.

[38] D. Wheeler. Flawfinder. `http://www.dwheeler.com/flawfinder`.

[39] J. Wilander and M. Kamkar. A comparison of publicly available tools for static intrusion prevention. In *Proc. of 7th Nordic Workshop on Secure IT Systems*, 2002.

[40] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proc. of 10th Network and Distributed System Security Symposium*, 2003.

[41] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient architectural support for software debugging. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004.