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

Alexey Smirnov and Tzi-cker Chiueh
SUNY at Stony Brook
{alexey, chiueh}@cs.sunysb.edu

DEFCON 13

# Introduction

- Buffer overflow attacks are the most common way for an attacker to gain control of a remote system.
- A comprehensive defense strategy should include of the following components:
  - *Attack detection* – to prevent the attack from causing damage and further propagation;
  - *Attack identification* – to prevent the attack from penetrating into the system in the future;
  - *Attack repair* – to allow the compromised application to continue its normal execution.
- In this presentation we describe a compile-time defense mechanism that provides all three components.

# Outline of the Talk

- Introduction
- Related Work
- DIRA Architecture
  - Attack Detection
  - Memory Logging Approach
  - Attack Identification (signature generation)
  - Attack Repair
  - Limitations
- Implementation Issues
- Performance Evaluation
- Conclusion

# What is a Buffer Overflow Attack

- Control-hijacking attacks work by overwriting a *control pointer* such as the return address, function pointer, stack frame pointer, jump table entry, interrupt handler address, etc.
- Buffer overflows are possible when the length of the target buffer is less than the length of the data that can be written into it.
- Standard *libc* functions such as *strcpy()* or *sprintf()* are responsible for most buffer overflows.
- Once the control is hijacked, it can be (1) redirected into the malicious code or (2) redirected into a standard function (*return-into-libc* attacks).

# Outline of the Talk

- Introduction
- Related Work
- DIRA Architecture
  - Attack Detection
  - Memory Logging Approach
  - Attack Identification and Repair
  - Limitations
- Implementation Issues
- Performance Evaluation
- Conclusion

# Taxonomy of Attack Detection Methods

- Extending the OS/hardware
  - Non-executable stacks and address-space randomization (PaX, Openwall for Linux, NGSEC StackDefender for Windows). Machine emulators (Bochs, Valgrind) allow to implement instruction randomization, pointer encryption, memory tainting.
- Extending the applications
  - Program analysis approaches: lint, Flawfinder, a number of commercial tools;
  - Run-time approaches: Libsafe, Libverify, Dinamo (program shepherding);
  - Hybrid approaches: program transformation + run-time monitoring – Stackguard, RAD.

# Source-code Based Attack Detection

- **Stackguard** – place a canary word before the RA in the function prologue and check it in the function epilogue. The assumption is that the attacker will have to overwrite the canary word in order to overwrite the RA.
- **RAD** – save the original RA in a safe place in the function prologue and compare it to the value stored in the stack in the function epilogue.

# Approaches to Attack Identification

- *Automatic* ways to identify attacks (that is, to generate their signatures) are very important for worm epidemics confinement.
- The previous systems either provided a single attacking packet or required a large pool of malicious network data.
- **Toth and Kruegel –** look at network packets payloads and perform abstract code execution.
- **TaintCheck –** uses the value of compromised control pointer as the attack signature.
- **Autograph –** extracts most common subsequences from suspicious flows and reports them as signatures.
- **Polygraph** and **Nemean –** use machine learning algorithms to derive common patterns from a large set of malicious flows.

# Approaches to Attack Repair

- Program rollback and replay is used in software debugging. Two approaches: (1) keep execution history (**Spyder**) or (2) do periodic state check-pointing. Check-pointing is easy under Linux because of copy-on-write *fork()* system call (**RECAP** and **Flashback**). Can be more difficult under other OS.
- Check-pointing relies on the OS rather than on the applications.
- **Shadow Honeypot** runs two versions of the application (protected and non-protected) and dynamically switches between the two once an attack has been detected.

# Outline of the Talk

- Introduction
- Related Work
- DIRA Architecture
  - Attack Detection
  - Memory Logging Approach
  - Attack Identification
  - Attack Repair
- Implementation Issues
- Performance Evaluation
- Conclusion

# DIRA Approach

- DIRA provides a unified compile-time solution to the three problems and is implemented as an extension to GCC 3.4.1. It uses a unified approach called *memory updates logging*.
- The idea is to maintain a run-time log of all operations that change the memory state of the program.
  - To **detect** an attack – compare the current RA with that saved in the log;
  - To **identify** the attack – trace back the data that replaced the control pointer to the point where this data first appeared in the program;
  - To **repair** the program – restore the memory state using the values stored in the log.
- DIRA has three modes of compilation: **D-mode** (detection only), **DI-mode** (detection+identification), **DIR-mode** (detection+identification +repair).
- At compile time, DIRA instruments the source code to perform logging and to check correctness of control pointers. At run-time, the logging code generates the memory updates log.

# Attack Detection (D-mode)

- DIRA uses RAD-like approach: the code to safe the RA in a protected buffer (for example, sliced between a pair of *mprotected()* pages) is added to the function prologue. The actual RA stored in the stack is compared with this value in function epilogue. Using a separate buffer to store RAs is an optimization of using a common memory update log to store RAs.
- DIRA can protect other sensitive data structures such as GOT, signal handler tables in a similar fashion (not implemented yet).

# Memory Updates Logging

- Memory updates log is a circular buffer; each entry has four fields: `read_addr`, `write_addr`, `len`, `data`.
- *Three sources* of memory image changes: assignment operations (X=Y) and standard functions (strcpy(), …), function invocations.
- DIRA logs effect of each operation of the form X=Y where X and Y are directly referenced variables, array references (a[i]), or de-referenced variables (*(a+1)).
  - `read_addr` is set to &Y,
  - `write_addr` is set to &X,
  - `len` is set to sizeof(Y),
  - `data` is set to the pre-image of X in DIR mode and is empty in other modes.

# Memory Updates Logging

- If the right-hand side is a complex expression then a log record is created for each variable of it.
- To handle updates performed by *libc functions* DIRA proxies a number of them.
  - String manipulation functions;
  - Format string functions;
  - File and Network I/O functions;
  - and others.
- The log is in fact a dual-purpose buffer. It is used to store the memory updates and *tags*, special marks indicating program state change.
  - FUNCTION_ENTRY tag is inserted when a function is called;
  - FUNCTION_EXIT tag is inserted before a function returns.

# Attack Identification (DI-mode)

- The purpose of attack detection is to identify attack packets and generate attack signatures that minimize the false positive and false negative rates.
- DIRA's signature format: *multiple packets*, each packet represented as a *regular expression*. The final attacking packet has the *length constraint* in it.
- Attack packet identification leverages the dependency information available in the memory updates log.
- There are two types of dependencies: data dependencies and control dependencies:
  - A *data dependency* is created whenever one variable is assigned to another.
  - A *control dependency* is created between a variable `X` and a variable `Y` when variable `X` used in a conditional expression can prevent control flow from reaching variable `Y`.

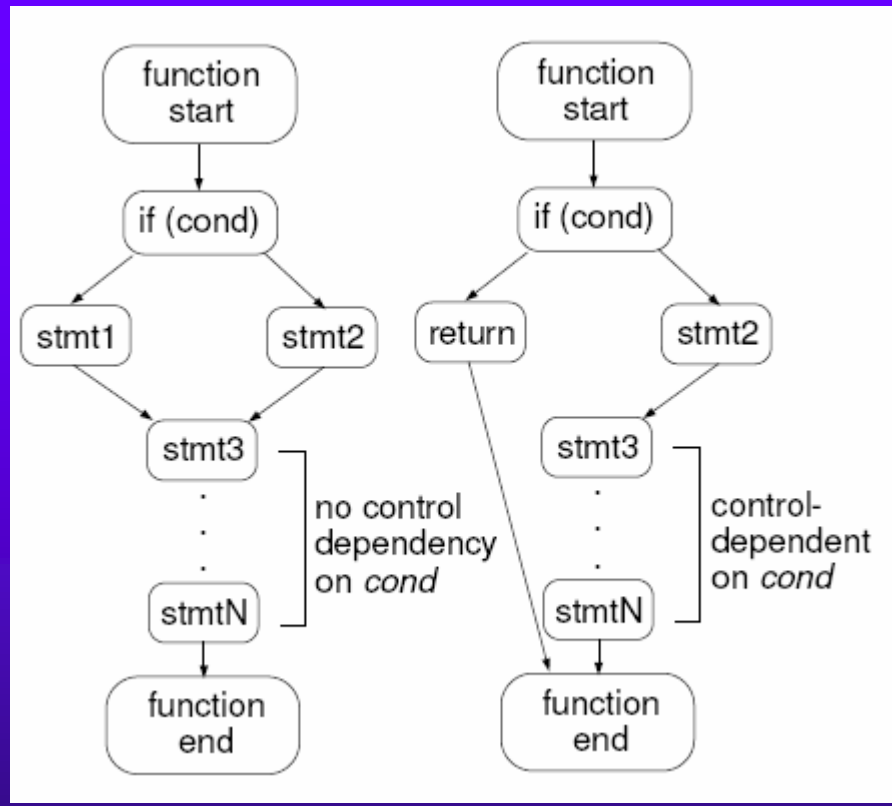
# Packet Identification Using Data Dependencies

- Trace-back algorithm allows one to trace the malicious data back to the point where it was received. The following code illustrates how data dependencies can be used to identify a malicious packet.

```
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 */
```

# Definition of Control Dependencies

- Whenever variable X can prevent control flow from reaching variable Y a control dependency between X and Y is created.
- stmt1 and stmt2 are always dependent.




- Control dependencies can also be created when loops such as **for** and **while** are used. These dependencies are also stored in the memory updates log.

# Complete Trace-back Algorithm

- Two special tags – START_SCOPE and END_SCOPE are added to the memory updates log. The conditional variables are stored in the START_SCOPE tag along with the closing scope ID.
- The new version of the trace-back algorithm will maintain a set of currently tracked addresses rather then just one address.
- In order to account for the control dependencies the trace-back algorithm should find all currently open scopes and add the variables saved in the START_SCOPE tags to the set of tracked addresses.
- Identifying all attack packets can reduce the false positives rate.

# Attack Identification (DI-mode)

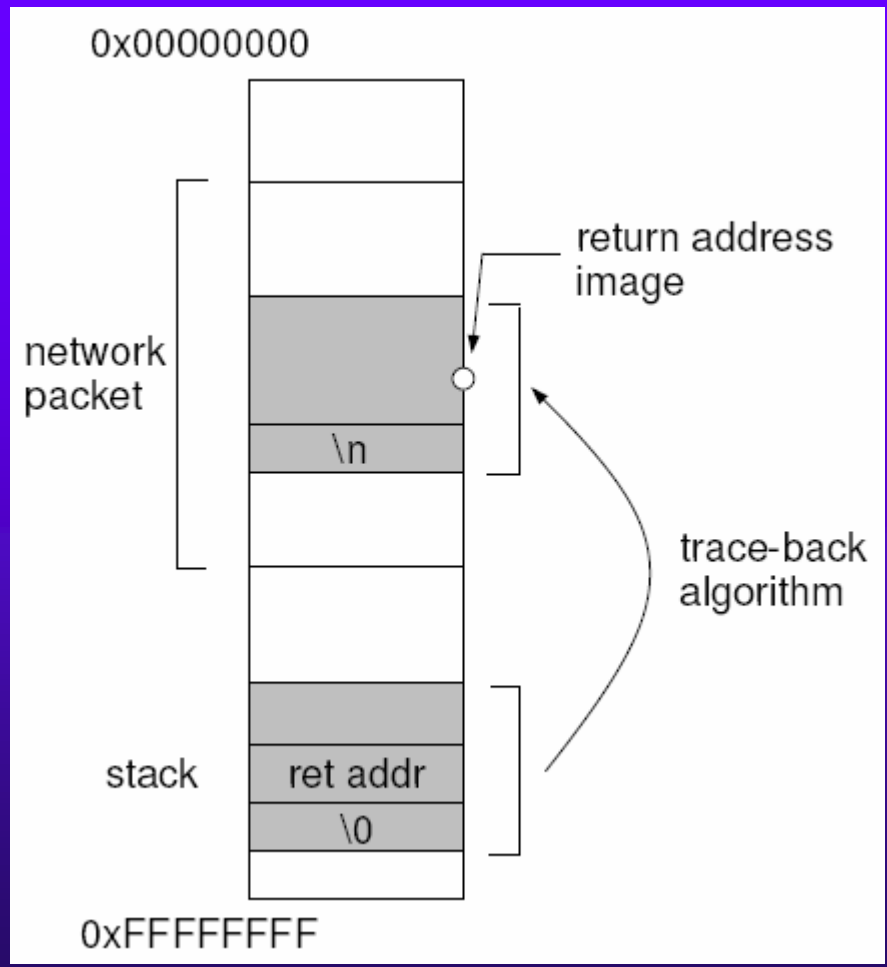
- Several classes of *libc* functions need to be proxied:
  - *Copying/concatenation* (*strcpy(a, b)*) – read_addr is set to the address of b, write_addr – address of a, len to strlen(b) and data to NULL (DI-mode);
  - *Network I/O* (*recv()*) – *read_addr*=-1 (external data). *data* field stores the post-image of the buffer being written; this data is presented as the malicious packet content when the trace-back algorithm finishes;
  - *File I/O* (*read()*) – same as network I/O;
  - *Format string* (*sprintf()*) – similar to copying functions, but can produce multiple log records;

# Representing Packets as Regular Expressions

- Polymorphic worms change their appearance from one attack instance to another so the packets need to be generalized.
- For each byte of the attacking packet DIRA determines whether it was *looked at* by the program or *not looked at*. For example, `strcmp()` applied to some bytes of the packet converts them into looked-at bytes. If, however, the bytes were blindly copied by a `strcpy()` then they are still non-looked-at.
- Initially all bytes are not-looked-at.
- DIRA traverses the log forward from where the packets were received and records all packet bytes that were looked at.

# Length Constraint Generation

- The length constraint limits the attacking part of the packet by specifying the terminating character and its maximum offset in any benign packet.



# DIRA's Signature File Format

- N – number of packets
- L_i – length of i-th packet
- Regular expression of the packet. Possible characters are shown on the right:

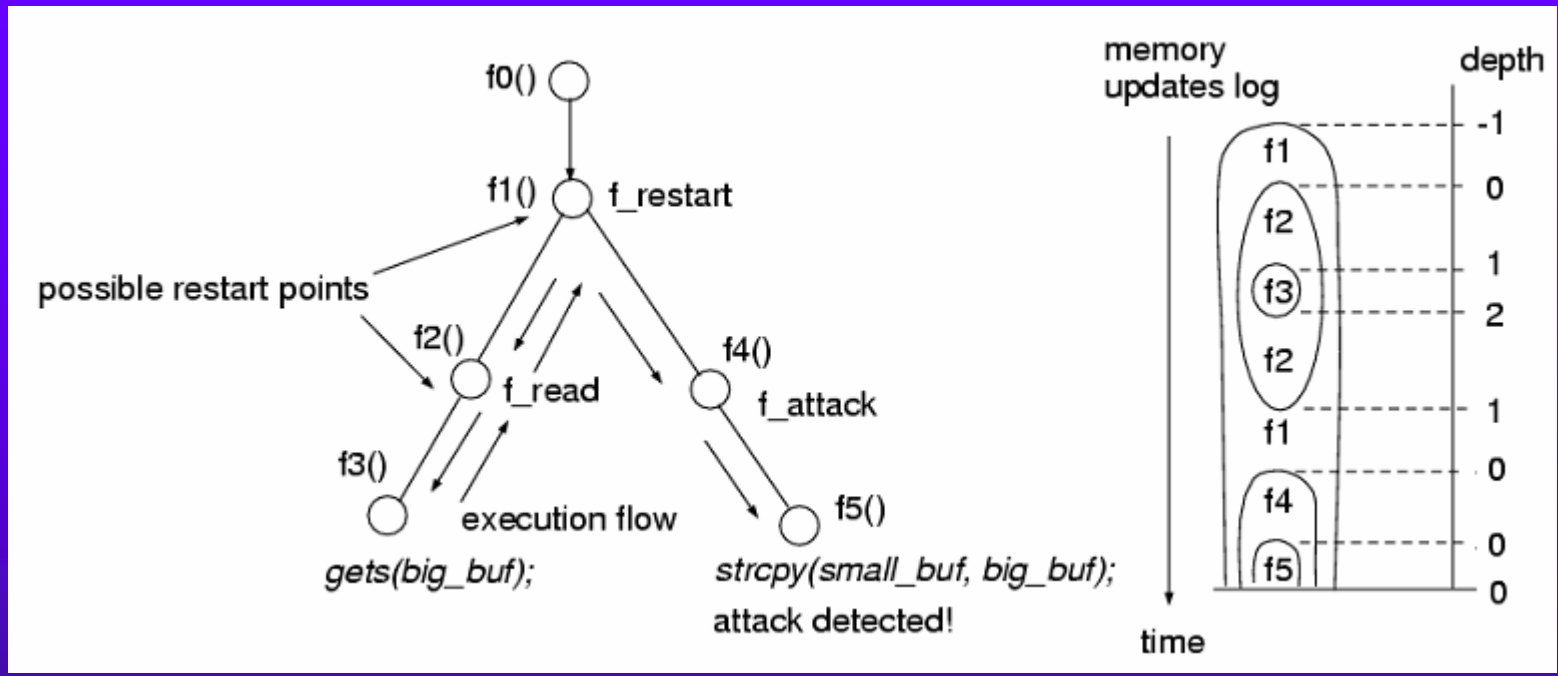
| Value | Description |
|---|---|
| ? | non-looked-at |
| \A | digit or letter |
| \a | letter |
| \I | ASCII character |
| \c | control character |
| \d | digit |
| \g | pseudo graphics |
| \l | lower case |
| \r | printable |
| \p | punctuation |
| \s | space |
| \u | upper |
| \x | hex digit |
| <value> | another looked-at |

- The length constraint is specified for the last attacking packet.

# Attack Recovery (DIR-mode)

- Main goal: bring the program to a state in which it was before the attack packet(s) was received.
- Two issues:
  1. How to restore the pre-attack state?
  2. From which point to continue execution?
- Memory updates log is used to solve (1). Program restart points can only be at the beginning of a function because only global updates are logged. The proper function turns out to be the *least common dynamic ancestor* of the function in which the attack was detected and the function in which the data was read in.

# Choosing the Restart Point




- **`depth`** is a *loop invariant*: it is the relative depth of the current function with respect to the greatest dynamic ancestor seen so far.

# Choosing the Restart Point

- Sometimes, it is possible to resume execution from the middle of a function. This becomes possible if there are no local variable updates between when f_read returns and f_attack begins.
- No system support is required for restarting – *longjmp()* and *setjmp()* are used. A *setjmp()* call is inserted before the function that can be a potential restart point is called (to push the arguments again).
- DIRA inserts the *first local update tag* when it encounters such an update after a function call.

# Tags Used in DIR-mode

- *Function entry tag* – inserted in function prologue;
- *Function exit tag* – inserted in function epilogue;
- *Jump buffer tag* – indicates that the data field contains data returned by *setjmp();*
- *First local update* – inserted when the first local update after a function call is encountered.

# Proxy Function for DIR-mode

- *Memory management functions* (malloc, free, …). Deferred free is used to keep the objects in memory as they can be brought to life again after repair completes. The data is actually freed when the log entry is revoked (circular buffer of limited length).
- *Interprocedural Jump Functions* (setjmp, longjmp) – proxy functions keep the log consistent by inserting the proper number of function exit tags.
- *Privilege Management Functions* (seteuid, setegid) – at repair time, the application needs to restore its original effective uid/gid.
- *Process Management Functions* (fork) - no need to create a new log explicitly because of copy-on-write semantics of fork. DIRA does not perform cascading rollback; it roll backs only the process in which an attack occurred. No change to parent process is performed.

# Limitations of the Prototype

- Single read address in logging. For B=A+C, the read address is set to ‘unknown’.
- Redundant logging. For example, if the same global variable is updated in a loop without any function calls, it needs to be logged only once.
- Better multithreading support needed (*vfork()*).
- Alarms and signals are not recovered.

# Outline of the Talk

- Introduction
- Related Work
- DIRA Architecture
  - Attack Detection
  - Memory Logging Approach
  - Attack Identification and Repair
  - Limitations
- Implementation Issues
- Performance Evaluation
- Conclusion

# Implementation Issues

- Source code instrumentation is performed at two levels: *AST level* and *machine code level*.
- Basic AST transformation: X=Y -> (log(&Y,&X,sizeof(Y),X), X=Y).
- Special care is taken of unary arithmetic operations (to avoid double modification).
- Each function call expression is prefixed with a call to setjmp(): func() -> (setjmp(), func()).
- Function prologue and epilogue are changed at machine code level (specific to GCC).

# Implementation Issues

- Special support for libraries: they should be reusable.
- Solution: each function is duplicated. The first copy is left intact, the second is instrumented. An if-expression is inserted in front of the function. It checks need_logging==0. If the condition is true then the uninstrumented branch is taken, otherwise the instrumented one is taken. need_logging is a special variable added to each library. It is set to zero by default and can be changed to one by any application that requires DIRA support.

# Outline of the Talk

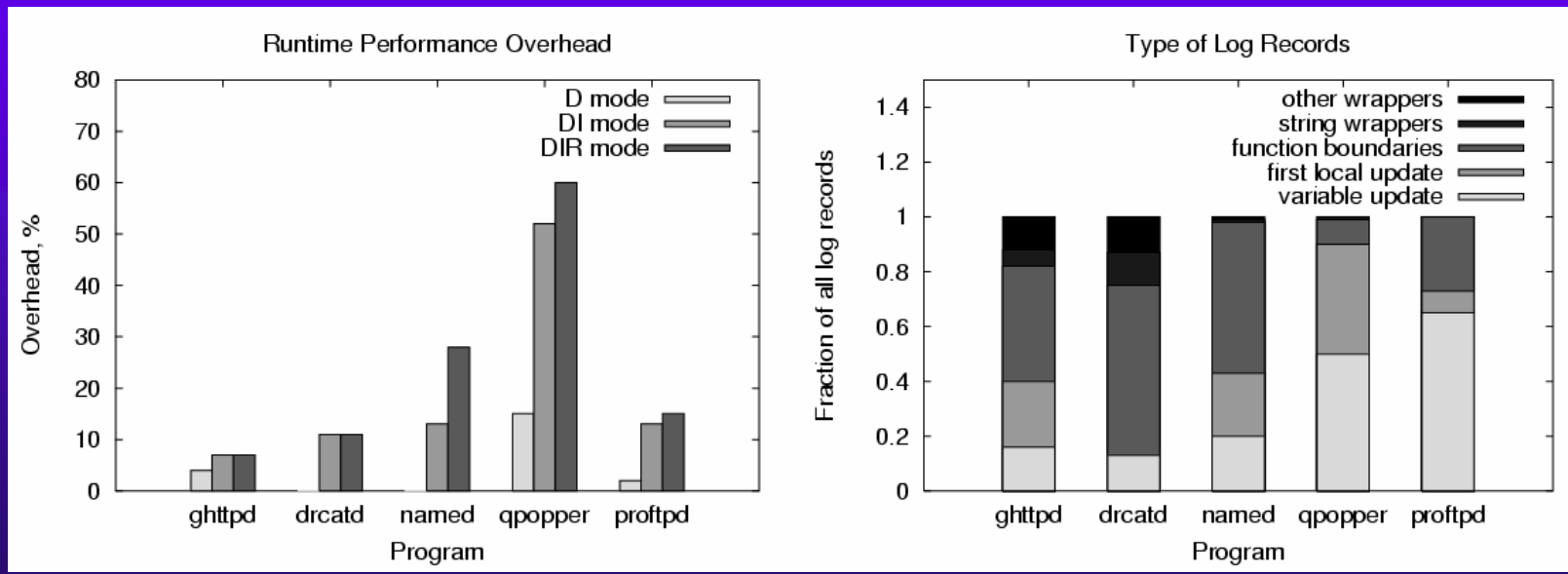
- Introduction
- Related Work
- DIRA Architecture
  - Attack Detection
  - Memory Logging Approach
  - Attack Identification and Repair
  - Limitations
- Implementation Issues
- Performance Evaluation
- Conclusion

# DIRA Evaluation

- Programs tested:
  - ghttpd 1.4 – have exploit;
  - drcatd 0.5.0 – have exploit;
  - named 8.1 – have exploit;
  - qpopper 4.0.4;
  - proftpd 1.2.9;
- Two goals: measure *run-time overhead* and study the impact of the recovery procedure on the *program availability*.
- Configuration: server machine (P-4M 1.7GHz, 512 MB RAM), two clients (Athlon 1.7GHz, 512 MB RAM).
- Used exploit programs from securiteam.com and insecure.org.

# Run-time Overhead

- The following two graphs show run-time overhead for programs compiled in DIR-mode:



# File System Undo

- Is it necessary? Three out of five programs of the test suite do not perform any file I/O at all. The remaining two write temp files and log information. This file system state is not critical.

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

# Program Recovery

- Bind 8.1 – recovery OK.
- Ghttpd – recovery OK.
- drcatd – not exactly OK. The least common dynamic ancestor is main(), and there are local updates between f_read and f_attack. Solutions (both non-automatic): (1) rewrite the source code so that the least common dynamic ancestor is not main (2) track all updates, not only locals. The second solution is used currently.

# Is Recovery Really Useful?

- Recovery does not work sometimes and can be expensive. Is it really better than just terminating the program? We believe that yes because:
  - In case of a single-threaded multi-client program (such as a high-performance web-server), terminating the program disconnects all clients.
  - This question is equivalent to asking whether the source-code checking tools are necessary or we can do well just by using Stackguard.

# Outline of the Talk

- Introduction
- Related Work
- DIRA Architecture
  - Attack Detection
  - Memory Logging Approach
  - Attack Identification and Repair
  - Limitations
- Implementation Issues
- Performance Evaluation
- Conclusion

# Conclusion

- DIRA is the first system that solves the problems of attack detection, identification, and repair in a unified way.
- It can produce accurate multi-packet signatures from a single attack instance.
- Our main research contribution is applying *dynamic slicing* which is a well-known technique in programming languages to some interesting security problems.
- We are planning to apply the same technique for automatic patch generation. Indeed, the information contained in the memory log allows one to tell the amount by which a buffer was overwritten and come up with a fix to the vulnerability such as using a larger buffer or limiting the number of bytes read from network.

# Questions?