Dynamic Detection of Address Leaks

Gabriel Silva Quadros, Rafael M. Souza and Fernando Magno Quintão Pereira

[1]Departamento de Ciência da Computação – UFMG
Av. Antônio Carlos, 6627 – 31.270-010 – Belo Horizonte – MG – Brazil

{gabrielquadros,rafaelms,fernando}@dcc.ufmg.br

Abstract. *An address leak is a software vulnerability that allows an adversary to discover where a program is loaded in memory. Although seemingly harmless, this information gives the adversary the means to circumvent two widespread protection mechanisms: Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP). In this paper we show, via an example, how to explore an address leak to take control of a remote server running on an operating system protected by ASLR and DEP. We then present a code instrumentation framework that hinders address disclosure at runtime. Finally, we use a static analysis to prove that parts of the program do not need to be instrumented; hence, reducing the instrumentation overhead. We claim in this paper that the combination of the static and dynamic analyses provide us with a reliable and practical way to secure software against address leaks.*

Resumo. *Um vazamento de endereço é uma vulnerabilidade de software que permite a um adversário descobrir em que parte da memória estão carregados os diversos módulos que compõem um programa. Embora aparentemente inócua, esse tipo de informação dá ao adversário meios para contornar dois populares mecanismos de proteção usados em sistemas operacionais: a randomização de espaços de endereçamento (ASLR) e a Prevenção de Execução de Dados (DEP). Neste artigo mostraremos, via um exemplo, como explorar vazamentos de endereço para tomar controle de um servidor remoto executando sobre um sistema operacional protegido tanto por ASLR quanto DEP. Mostraremos em seguida um arcabouço para instrumentação de programas que previne vazamentos em tempo de execução. Finalmente, nós usaremos uma análise estática de código que prova que algumas partes do programa não precisam ser instrumentadas para reduzir o custo imposto pela instrumentação. Defendemos assim, neste artigo a tese de que a combinação de análises estáticas e dinâmicas é um recurso efetivo e prático para proteger programas contra o vazamento de endereços.*

1. Introduction

Modern operating systems use a protection mechanism called *Address Space Layout Randomization* (ASLR) [Bhatkar et al. 2003, Shacham et al. 2004]. This technique consists in loading the binary modules that form an executable program at different addresses each time the program is executed. This security measure protects the software from well-known attacks, such as *return-to-libc* [Shacham et al. 2004] and *return-oriented-programming* (ROP) [Buchanan et al. 2008, Shacham 2007]. Because it is effective, and easy to implement, ASLR is present in virtually every contemporary operating system. However, this kind of protection is not foolproof.

Shacham *et al.* [Shacham et al. 2004] have shown that address obfuscation methods are susceptible to brute force attacks; nevertheless, address obfuscation slows down the propagation rate of worms that rely on buffer overflow vulnerabilities substantially. However, an adversary can still perform a surgical attack on an ASLR protected program. In the words of the original designers of the technique [Bhatkar et al. 2003, p.115], if "the program has a bug which allows an attacker to read the memory contents", then "the attacker can craft an attack that succeeds deterministically". It is this very type of bug that we try to prevent in this paper.

In this paper we focus on dynamic techniques to prevent address leaks. We have developed an instrumentation framework that automatically converts a program into a software that cannot contain address leaks. This transformation, which we introduce in Section 2, consists in instrumenting every program operation that propagates data. In this way, we know which information might contain address knowledge, and which information might not. If harmful information reaches an output point that an adversary can read, the program stops execution. Thus, by running the instrumented, instead of the original software, the user makes it much harder for an adversary to perform attacks that require address information to succeed.

Non-surprisingly, the instrumentation imposes on the target program a very large runtime overhead: the sanitized program can be as much as 10x slower than the original code. However, we use a static program analysis to reduce this overhead. As we explain in Section 2.1, we only instrument the pieces of code that can deal with potentially harmful data. Innocuous operations, which the static analysis identifies, are not instrumented. We are currently using the static analysis that we had previously designed and implemented to detect address disclosure vulnerabilities [Quadros and Pereira 2011]. The combination of static analysis and dynamic instrumentation lets us produce code secured against address leaks that is, on the average, less than 50% slower than the original code.

We have implemented our instrumentation framework, and the companion static analysis in the LLVM compiler [Lattner and Adve 2004]. The experiments that we describe in Section 3 show encouraging results. We have used our tool to analyze a large number of programs, and in this paper we show results for two well-known benchmark collections: FreeBench and Shootout. Instrumented programs can be 2% to 1,070% slower than the original programs, with an average slowdown of 490%. The static analysis contributes substantially to decrease this overhead. By only instrumenting the operations that the static analysis has not been able to prove safe, we get slowdowns ranging from 0% to 420%, with an average slowdown of 44%.

1.1. Address Leak in one Example

We illustrate the address disclosure vulnerability via the echo server seen in Listing 1. The information leak in this example let us perform a stack overflow attack on a 32-bit machine running Ubuntu 11.10, an operating system protected by ASLR and DEP. In this example, the vulnerable program keeps listening for clients at port 4000, and when a client connects, it echoes every data received. DEP hinders a buffer overflow attach in the classic Levy style [Levy 1996], because it forbids the execution of writable memory space. However, we can use a buffer overflow to divert program execution to one of `libc`'s functions. Such functions allows us to fork new processes, to send e-mails and to open socket connections, for instance. In this example we shall open a telnet terminal in the server's machine. This type of attack, usually called return-to-libc, depends on the adversary knowing the address of the target function, e.g., `libc`'s `system` in this example. This information is not easily available in a ASLR protected system, unless the target software contains an address leak.

Listing 1. An echo server that contains an address leak.

```
void *libc;
void process_input(char *inbuf, int len, int clientfd) {
    char localbuf[40];
    if (!strcmp(inbuf, "debug\n")) {
        sprintf(localbuf, "localbuf␣%p\nsend()␣%p\n", localbuf,
                dlsym(libc, "send"));
    } else { memcpy(localbuf, inbuf, len); }
    send(clientfd, localbuf, strlen(localbuf), 0);
}
int main() {
    int sockfd, clientfd, c_len, len;
    char inbuf[5001];
    struct sockaddr_in myaddr, addr;
    libc = dlopen("libc.so", RTLD_LAZY);
    sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    myaddr.sin_family = AF_INET;
    myaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    myaddr.sin_port = htons(4000);
    bind(sockfd, (struct sockaddr *)&myaddr, sizeof(myaddr));
    listen(sockfd, 5);
    c_len = sizeof(addr);
    while (1) {
        clientfd = accept(sockfd, (struct sockaddr *)&addr, &c_len);
        len = recv(clientfd, inbuf, 5000, 0);
        inbuf[len] = '\0';
        process_input(inbuf, len + 1, clientfd);
        close(clientfd);
    }
    close(sockfd); dlclose(libc);
    return 0;
}
```

The information leak in our example occurs at function `process_input`. Whenever the server recognizes the special string "debug" it returns two internal addresses: the base of `localbuf`, which is a stack address, and the address of `send`, a function from `libc`. To build the exploit, we use the address of `send` to calculate the address of `system` and `exit`, two functions present in `libc`. We then use the stack address of `localbuf`'s base pointer, to find the address of `system`'s arguments. A Python script that performs this exploit is shown in Listing 2. This script makes two connections to the echo server. In the first connection it sends the string "debug" to read back the two leaked addresses. In the second connection it sends the malicious data to create a connect-back shell. The malicious data is composed of: 52 A's to fill the stack until before the return pointer; the address of `system`, calculated from the leaked address of `send`;

the address of `exit`, also computed from the address of `send`; the address of the string with the command to create the connect-back shell, calculated from `localbuf`'s base pointer; and finally the string containing the command to create the shell. By overwriting the return address of `process_input` with the address of `system` we gain control of the remote machine. By calling `exit` at the end of the exploit we ensure that our client terminates quietly after giving us a shell.

Listing 2. A Python script that exploits the echo server.

```
import socket
import struct
c = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
c.connect(('localhost', 4000))
buf = "debug\n"
c.send(buf)
buf = c.recv(512)
leaked_stack_addr = int(buf[9:buf.find('\n')], 16)
leaked_send_addr = int(buf[27:buf.rfind('\n')], 16)
c.close()
c = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
c.connect(('localhost', 4000))
command = ('rm␣-f␣backpipe␣&&␣mknod␣backpipe␣p␣&&'
        'telnet␣localhost␣8080␣0<backpipe␣|␣/bin/bash␣1>backpipe\x00')
command_addr = leaked_stack_addr + 64
system_addr = leaked_send_addr - 0x96dd0  # system()
system_ret_addr = system_addr - 0xa140 # exit()
buf = ('A' * 52 + struct.pack('I', system_addr) +
        struct.pack('I', system_ret_addr) +
        struct.pack('I', command_addr) + command)
c.send(buf)
c.close()
```

2. The Proposed Solution

In this section we describe our solution to detect address leaks at runtime. We start by defining a core language containing the constructs from imperative languages that play a role in the address disclosure vulnerability. On top of this language we define an *instrumentation language*. Programs implemented with the instrumentation syntax can track the flow of information at runtime. We then proceed to describe a type system that detects address leaks statically. This type system let us reduce the amount of instrumentation necessary to safe-guard programs against address leaks.

Angels: the Subject Language. We define a toy language, which we call *Angels*, to explain our approach to dynamic detection of address leaks. Angels is an assembly-like language, whose syntax is given in Figure 1. This language has six instructions that deal with the computation of data, and three instructions that change the program flow. The six data related instructions represent constructs typical of actual C or C++ programs, as the table in Figure 2 illustrates. We use `adr` to model language constructs that read the address of a variable, namely the ampersand (&) operator and memory allocation functions such as `malloc`, `calloc` or `realloc`. Simple assignments are represented via the instruction `mov`. We represent binary operations via the `add` instruction, which sums up two variables and dumps the result into a third location. Loads to and stores from memory

(Variables)	::=	$\{v_1, v_2, \ldots\}$
(Data Producing Instructions)	::=	
– (Get variable's address)	$\mid$	adr(v_1, v_2)
– (Assign to variable)	$\mid$	mov(v_1, v_2)
– (Binary addition)	$\mid$	add(v_1, v_2, v_3)
– (Store into memory)	$\mid$	stm(v_0, v_1)
– (Load from memory)	$\mid$	ldm(v_1, v_0)
– (Print the variable's value)	$\mid$	out(v)
(Control flow Instructions)	::=	
– (Branch if zero)	$\mid$	bzr(v, l)
– (Unconditional jump)	$\mid$	jmp(l)
– (Halt execution)	$\mid$	end
(ϕ-function - data selector)	$\mid$	phi($v, \{v_1 : l_1, \ldots, v_k : l_k\}$)

Figure 1. The syntax of Angels.

are modeled by `ldm` and `stm`. Finally, we use `out` to denote any instruction that gives information away to an external user. This last instruction represents not only ordinary printing operations, but also native function interfaces. For instance, a JavaScript program usually relies on a set of native functions to interact with the browser. A malicious user could use this interface to obtain an internal address from the JavaScript interpreter.

We work with programs in the Static Single Assignment (SSA) form [Cytron et al. 1991]. This intermediate representation has the key property that every variable name has only one definition site in the program code. To ensure this invariant, the SSA intermediate representation uses ϕ-functions, a special notation that does not exist in usual assembly languages. Figure 1 shows the syntax of ϕ-functions. This representation is not necessary for computational completeness; however, as we will see in Section 2.1, it simplifies the static analysis of programs. Additionally, the SSA format is used by our baseline compiler, LLVM, and many other modern compilers, including gcc. Thus, by adopting this representation we shrink the gap between our abstract formalism and its concrete implementation.

Figure 3 describes the small-steps operational semantics of Angels. We let an abstract machine be a five-element tuple $\langle P, \texttt{pc}', \texttt{pc}, \Sigma, \Theta\rangle$. We represent the program P as a map that associates integer values, which we shall call *labels*, with instructions. We let `pc` be the current *program counter*, and we let `pc`' be the program counter seen by the last instruction processed. We need to keep track of the previous program counter to model ϕ-functions. These instructions are used like variable multiplexers. For instance, $v = \phi(v_1 : l_1, v_2 : l_2)$ copies v_1 to v if control comes from l_1, and copies v_2 to v if control comes from l_2. The previous program counter points out to us the path used to reach the ϕ-function. Σ, the memory, is an environment that maps variables to integer values, and Θ is an output channel. We use the notation $f[a \mapsto b]$ to denote the updating of function f; that is, $\lambda x.x = a \ ? \ b : f(x)$. For simplicity we do not distinguish a memory of local variables, usually called *stack*, from the memory of values that out-live the functions that have created them, usually called *heap*. We use a function Δ to map the names of

`v1 = &v2`	adr(v_1, v_2)
`v1 = (int*)malloc(sizeof(int))`	adr(v_1, v_2) where v_2 is a fresh memory location
`v1 = *v0`	ldm(v_1, v_0)
`*v0 = v1`	stm(v_0, v_1)
`*v1 = *v0`	ldm(v_2, v_0), where v_2 is fresh stm(v_1, v_2)
`v1 = v2 + v3`	add(v_1, v_2, v_3)
`*v = v1 + &v2`	adr(v_3, v_2), where v_3 is fresh add(v_4, v_1, v_3), where v_4 is fresh stm(v, v_4)
`f(v1, &v3)`, where `f` is declared as `f(int v2, int* v4);`	mov(v_2, v_1) adr(v_4, v_3)

Figure 2. Examples of syntax of C mapped to instructions of Angels.

variables to their integer addresses in Σ. We represent the output channel as a list Θ. As we see in Rule [OUTSEM], the only instruction that can manipulate this list, by *consing* a value on it, is the `out` instruction. We denote consing by the operator ::, as in ML and Ocaml.

Angels is a Turing Complete programming language. Given an infinite surplus of variables, we can implement a Turing Machine on it. Figure 4 shows an example of a program written in Angels. The program in Figure 4(a) prints the contents of an array, and then the base address of the array itself. Figure 4(b) shows the equivalent Angels program. We have outlined the label of the first instruction present in each basic block of this example. We have also marked in gray the initial memory used in this example. Angels does not have instructions to load constants into variables; however, we compensate this omission by starting programs with a non-empty memory.

The program in Figure 4 contains an address disclosure vulnerability, a notion that we introduce in Definition 2.1. An Angels program P contains such a vulnerability if an adversary can reconstruct the map Δ for at least one variable that P uses. Notice that we are not assuming that P terminates. We only require that the program outputs any information that an adversary can read. Our running example contains an address leak vulnerability, because it prints the base address of the array f. Thus, by reading the output channel, the adversary would be able to find $\Delta(f)$.

Definition 2.1 THE ADDRESS DISCLOSURE VULNERABILITY. *An Angels program P, such that $\langle P, 0, \lambda x.0, [] \rangle \rightarrow \langle \Sigma, \Theta \rangle$, where $\lambda x.0$ is the environment that maps every variable to zero, and $[]$ is the empty output channel, contains an address disclosure vulnerability if it is possible for an adversary to discover $\Delta(v)$ for some v used in P, given the knowledge of Θ plus the source code of P.*

The Instrumented language. In order to secure a program against address disclosures,

$$[\textsc{AdrSem}]\quad \frac{\Delta(v_2) = n \qquad \Sigma' = \Sigma[\Delta(v_1) \mapsto n]}{\langle \texttt{adr}(v_1, v_2), \Sigma, \Theta\rangle \to \langle \Sigma', \Theta\rangle} \qquad [\textsc{OutSem}]\quad \frac{\Sigma[\Delta(v)] = n \qquad \Theta' = n :: \Theta}{\langle \texttt{out}(v), \Sigma, \Theta\rangle \to \langle \Sigma, \Theta'\rangle}$$

$$[\textsc{MovSem}]\quad \frac{\Sigma[\Delta(v_2)] = n \qquad \Sigma' = \Sigma[\Delta(v_1) \mapsto n]}{\langle \texttt{mov}(v_1, v_2), \Sigma, \Theta\rangle \to \langle \Sigma', \Theta\rangle}$$

$$[\textsc{AddSem}]\quad \frac{\Sigma[\Delta(v_2)] = n_2 \qquad \Sigma[\Delta(v_3)] = n_3 \qquad \Sigma' = \Sigma[\Delta(v_1) \mapsto n_2 + n_3]}{\langle \texttt{add}(v_1, v_2, v_3), \Sigma, \Theta\rangle \to \langle \Sigma', \Theta\rangle}$$

$$[\textsc{StmSem}]\quad \frac{\Sigma[\Delta(v_0)] = x \qquad \Sigma[\Delta(v_1)] = n \qquad \Sigma' = \Sigma[x \mapsto n]}{\langle \texttt{stmem}(v_0, v_1), \Sigma, \Theta\rangle \to \langle \Sigma', \Theta\rangle}$$

$$[\textsc{LdmSem}]\quad \frac{\Sigma[\Delta(v_0)] = x \qquad \sigma[x] = n \qquad \Sigma' = \Sigma[\Delta(v_1) \mapsto n]}{\langle \texttt{ldmem}(v_1, v_0), \Sigma, \Theta\rangle \to \langle \Sigma', \Theta\rangle}$$

$$[\textsc{EndSem}]\quad \frac{P[\mathrm{pc}] = \texttt{end}}{\langle P, \mathrm{pc}', \mathrm{pc}, \Sigma, \Theta\rangle \to \langle \Sigma, \Theta\rangle}$$

$$[\textsc{JmpSem}]\quad \frac{P[\mathrm{pc}] = \texttt{jmp}(l) \qquad \langle P, \mathrm{pc}, l, \Sigma, \Theta\rangle \to \langle \Sigma', \Theta'\rangle}{\langle P, \mathrm{pc}', \mathrm{pc}, \Sigma, \Theta\rangle \to \langle \Sigma', \Theta'\rangle}$$

$$[\textsc{BzrSem}]\quad \frac{P[\mathrm{pc}] = \texttt{bzr}(v, l) \qquad \Sigma[\Delta(v)] \neq 0 \qquad \langle P, \mathrm{pc}, \mathrm{pc}+1, \Sigma, \Theta\rangle \to \langle \Sigma', \Theta'\rangle}{\langle P, \mathrm{pc}', \mathrm{pc}, \Sigma, \Theta\rangle \to \langle \Sigma', \Theta'\rangle}$$

$$[\textsc{BnzSem}]\quad \frac{P[\mathrm{pc}] = \texttt{bzr}(v, l) \qquad \Sigma[\Delta(v)] = 0 \qquad \langle P, \mathrm{pc}, l, \Sigma, \Theta\rangle \to \langle \Sigma', \Theta'\rangle}{\langle P, \mathrm{pc}', \mathrm{pc}, \Sigma, \Theta\rangle \to \langle \Sigma', \Theta'\rangle}$$

$$[\textsc{PhiSem}]\quad \frac{\begin{array}{c} P[\mathrm{pc}] = \texttt{phi}(v, \{v_1 : l_1, \ldots, v_k : l_k\}) \\ \mathrm{pc}' = l_i \qquad \langle \texttt{mov}(v, v_i), \Sigma, \Theta\rangle \to \langle \Sigma', \Theta\rangle \qquad \langle P, \mathrm{pc}, \mathrm{pc}+1, \Sigma', \Theta\rangle \to \langle \Sigma'', \Theta'\rangle \end{array}}{\langle P, \mathrm{pc}', \mathrm{pc}, \Sigma, \Theta\rangle \to \langle \Sigma'', \Theta'\rangle}$$

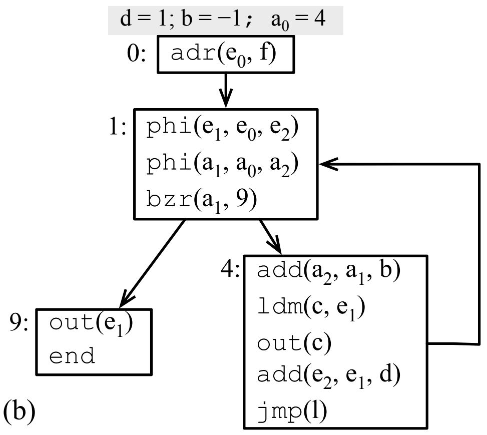
$$[\textsc{SeqSem}]\quad \frac{P[\mathrm{pc}] \notin \{\texttt{bzr}, \texttt{end}, \texttt{phi}, \texttt{jmp}\} \qquad \langle P[\mathrm{pc}], \Sigma, \Theta\rangle \to \langle \Sigma', \Theta'\rangle \qquad \langle P, \mathrm{pc}, \mathrm{pc}+1, \Sigma', \Theta'\rangle \to \langle \Sigma'', \Theta''\rangle}{\langle P, \mathrm{pc}', \mathrm{pc}, \Sigma, \Theta\rangle \to \langle \Sigma'', \Theta''\rangle}$$

Figure 3. The Operational Semantics of Angels.

(a)

```
int main() {
    int i = 0, f[4];
    for (i = 0; i < 4; i++) {
        printf("%d\n", f[i]);
    }
    printf("%u\n", f);
}
```

(b)



```
d = 1; b = -1;  a0 = 4
0: adr(e0, f)
1: phi(e1, e0, e2)
   phi(a1, a0, a2)
   bzr(a1, 9)
4: add(a2, a1, b)
   ldm(c, e1)
   out(c)
   add(e2, e1, d)
   jmp(l)
9: out(e1)
   end
```

Figure 4. A C program translated to Angels.

we will instrument it. In other words, we will replace its original sequence of instructions by other instructions, which track the flow of values at runtime. It is possible to instrument an Angels program using only Angels instructions. However, to perform the instrumen-

[ADRINS] $$\frac{\langle \texttt{adr}, \Sigma, \Theta\rangle \to \langle \Sigma', \Theta\rangle \qquad \Delta(v_1) = n_1 \qquad \Sigma'' = \Sigma'[n_1 + D \mapsto \mathit{tainted}]}{\langle \texttt{sh\_adr}(v_1, v_2), \Sigma, \Theta\rangle \to \langle \Sigma'', \Theta\rangle}$$

[OUTINS] $$\frac{\Delta(v) = n_v \qquad \Sigma[n_v + D \mapsto \mathit{clean}] \qquad \langle \texttt{out}(v), \Sigma, \Theta\rangle \to \langle \Sigma, \Theta'\rangle}{\langle \texttt{sh\_out}(v), \Sigma, \Theta\rangle \to \langle \Sigma, \Theta'\rangle}$$

[MOVINS] $$\frac{\langle \texttt{mov}(v_1, v_2), \Sigma, \Theta\rangle \to \langle \Sigma', \Theta\rangle \qquad \Sigma'[\Delta(v_2) + D] = t \qquad \Sigma'' = \Sigma'[\Delta(v_1) + D \mapsto t]}{\langle \texttt{sh\_mov}(v_1, v_2), \Sigma, \Theta\rangle \to \langle \Sigma'', \Theta\rangle}$$

[ADDINS] $$\frac{\begin{array}{c}\langle \texttt{add}(v_1, v_2, v_3), \Sigma, \Theta\rangle \to \langle \Sigma', \Theta\rangle \\ \Sigma'[\Delta(v_2) + D] = t_2 \qquad \Sigma'[\Delta(v_3) + D] = t_3 \qquad t_1 = t_2 \sqcap_{sh} t_3 \qquad \Sigma'' = \Sigma[\Delta(v_1) + D \mapsto t]\end{array}}{\langle \texttt{sh\_add}(v_1, v_2, v_3), \Sigma, \Theta\rangle \to \langle \Sigma'', \Theta\rangle}$$

[STMINS] $$\frac{\Sigma[\Delta(v_0)] = x \qquad \Sigma[\Delta(v_1)] = n \qquad \Sigma' = \Sigma[x \mapsto n] \qquad \Sigma'[\Delta(v_1) + D] = t \qquad \Sigma'' = \Sigma'[x + D \mapsto t]}{\langle \texttt{sh\_stm}(v_0, v_1), \Sigma, \Theta\rangle \to \langle \Sigma'', \Theta\rangle}$$

[LDMINS] $$\frac{\Sigma[\Delta(v_0)] = x \qquad \Sigma[x] = n \qquad \Sigma' = \Sigma[\Delta(v_1) \mapsto n] \qquad \Sigma'[x + D] = t \qquad \Sigma'' = \Sigma'[\Delta(v_1) + D \mapsto t]}{\langle \texttt{sh\_ldm}(v_1, v_0), \Sigma, \Theta\rangle \to \langle \Sigma'', \Theta\rangle}$$

[PHIINS] $$\frac{\begin{array}{c}P[\texttt{pc}] = \texttt{sh\_phi}(v, \{v_1 : l_1, \ldots, v_k : l_k\}) \\ \texttt{pc}' = l_i \qquad \langle \texttt{sh\_mov}(v, v_i), \Sigma, \Theta\rangle \to \langle \Sigma', \Theta\rangle \qquad \langle P, \texttt{pc}, \texttt{pc} + 1, \Sigma', \Theta\rangle \to \langle \Sigma'', \Theta'\rangle\end{array}}{\langle P, \texttt{pc}', \texttt{pc}, \Sigma, \Theta\rangle \to \langle \Sigma'', \Theta'\rangle}$$

Figure 5. The Operational Semantics of the instrumented instructions.

tation in this way we would have to make this presentation unnecessarily complicated. To avoid this complexity we define a second set of instructions, whose semantics is given in Figure 5. The instrumentation framework defines an equivalent shadow instruction for `out` and for each instruction that can update the memory Σ. To hold the meta-data produced by the instrumentation we create a *shadow memory*. For each variable v, stored at $\Sigma[\Delta(v)]$, we create a new location $\Sigma[\Delta(v) + D]$, where D is a displacement that separates each variable from its shadow. In other words, information about shadow and original variables are kept into the same store. The shadow values can be bound to one of two values, *clean* or *tainted*. Instrumented programs are evaluated by the same rules seen in Figure 3, augmented with the six new rules given in Figure 5. It is important to notice that the instrumented programs might terminate prematurely. Rule OUTINS does not progress if we try to print a variable that is shadowed with the tainted value. The meet operator used in the definition of `sh_add`, $\sqcap_{sh}$ is such that $n_1 \sqcap_{sh} n_2 = \mathit{tainted}$ whenever one of them is *tainted*, and is clean otherwise.

Figure 6 defines a relation ι that converts an ordinary Angels program into an instrumented program. Every instruction that can store data into the memory is instrumented; thus, we call this relation the *full instrumentation framework*. Notice that the instructions that control the execution flow are not instrumented. They do not need to be instrumented because neither of them generates new data in the store.

Theorem 2.2 *Let P be an angels program, and P^ι be such that $P \xrightarrow{\iota} P^\iota$. Let $I \in P$ and $I^\iota \in P^\iota$ such that $I \xrightarrow{\iota} I^\iota$. If a variable v is used at an instruction $I \in P$, and v is data dependent on some address, then $\Sigma[\Delta(v) + D]$ contains the value* tainted *when v is used at I^ι.*

Theorem 2.3 *If P is an angels program, and P^ι is such that $P \xrightarrow{\iota} P^\iota$, then $\langle P^\iota, 0, 0, \lambda x.0, []\rangle$ does not contain an address disclosure vulnerability.*

$$\begin{array}{rclrcl}
\texttt{adr}(v_1, v_2) & \xrightarrow{\iota} & \texttt{sh\_adr}(v_1, v_2) & \texttt{stm}(v_0, v_1) & \xrightarrow{\iota} & \texttt{sh\_stm}(v_0, v_1) \\
\texttt{out}(v) & \xrightarrow{\iota} & \texttt{sh\_out}(v) & \texttt{ldm}(v_0, v_1) & \xrightarrow{\iota} & \texttt{sh\_ldm}(v_0, v_1) \\
\texttt{mov}(v_1, v_2) & \xrightarrow{\iota} & \texttt{sh\_mov}(v_1, v_2) & \texttt{bzr}(v, l) & \xrightarrow{\iota} & \texttt{bzr}(v, l) \\
\texttt{add}(v_1, v_2, v_3) & \xrightarrow{\iota} & \texttt{sh\_add}(v_1, v_2, v_3) & \texttt{jmp}(l) & \xrightarrow{\iota} & \texttt{jmp}(l) \\
\texttt{phi}(v, \{.., v_i : l_i, ..\}) & \xrightarrow{\iota} & \texttt{sh\_phi}(v, \{.., v_i : l_i, ..\}) & \texttt{end} & \xrightarrow{\iota} & \texttt{end}
\end{array}$$

Figure 6. The full instrumentation of an angels program.

2.1. Using Static Analysis to Reduce the Instrumentation Overhead.

As we will show empirically, a fully instrumented program is considerably slower than the original program. In order to decrease this overhead, we couple the dynamic instrumentation with a static analysis that detects address leaks. We have developed this static analysis in a previous work [Quadros and Pereira 2011]. This static analysis is a form of tainted flow analysis [Schwartz et al. 2010]. Thus, the objective of the static analysis is to find a path from a source of sensitive data, i.e., an `adr` instruction, to a output channel, i.e., an `out` instruction. Paths are formed by *dependency relations*. If a variable v_1 is defined by an instruction i that uses a variable v_2, then we say that v_1 depends on v_2. As an example, v_1 depends on v_2 if the instruction $\texttt{mov}(v_1, v_2)$ is in the program.

We call the graph formed by all the dependencies in the program the *dependence graph*. We say that a variable v is *tainted* if: (i) v transitively depends on some variable v_1 created by an `adr` instruction; and (ii) there is some variable v_2 used in an `out` instruction that depends on v. Our static analysis [Quadros and Pereira 2011] determines the set of program variables that are tainted.

Partial Instrumentation. Given the result of our static analysis, we can reduce the amount of instrumented code in the target program that the ι relation from Figure 6 inserts. With such purpose, we define a new instrumentation framework as the γ relation, which is given in Figure 7. Contrary to the algorithm in Figure 6, this time we only change an instruction if (i) it defines a variable that has the *tainted* type; or (ii) it is the output operation, and uses a tainted variable. Theorem 2.4 proves that the partial instrumentation is correct. For the sake of space, we omit from Figure 7 the rules that do not change the target instruction. An instruction might remain unchanged in the instrumented program, either because this instruction has been deemed safe by the static analysis, or because it does not create values in memory.

Theorem 2.4 *If P is an angels program, and P^γ is such that $P \xrightarrow{\gamma} P^\gamma$, then $\langle P^\gamma, 0, 0, \lambda x.0, [\,]\rangle$ does not contain an address leak.*

Figure 8 shows the result of applying the static analysis onto the program seen in Figure 4(b). Figure 8(a) shows the dependence graph that the target program induces. Each variable in this graph has been annotated with the abstract state that our static analysis infers to it. Thus, a variable may have been assigned the type *clean* (C) or *tainted* (T). In this case, the variables e_0, e_1 and e_2 have been marked *tainted* (T). The partial instrumentation framework changes the instructions that either define these variables, or output them. Figure 8(b) shows the result of this transformation. The figure shows that only four, out of 11 instructions have been instrumented. This partially instrumented program is certainly more efficient than the fully instrumented program.

$$\Gamma \vdash v_1 = \mathit{tainted} \quad \Rightarrow \quad \langle \mathtt{adr}(v_1, v_2), \Gamma, \Pi\rangle \xrightarrow{\gamma} \mathtt{sh\_adr}(v_1, v_2)$$

$$\Gamma \vdash v = \mathit{tainted} \quad \Rightarrow \quad \langle \mathtt{out}(v), \Gamma, \Pi\rangle \xrightarrow{\gamma} \mathtt{sh\_out}(v)$$

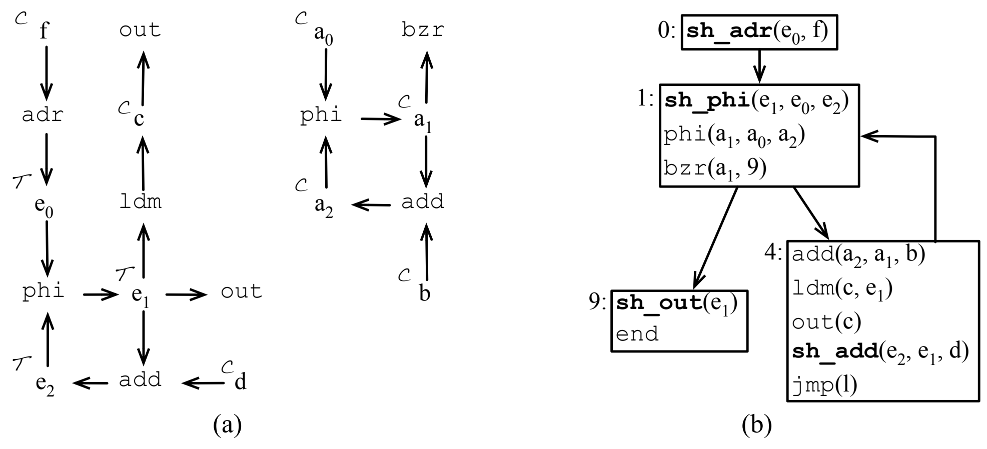
$$\Gamma \vdash v_1 = \mathit{tainted} \quad \Rightarrow \quad \langle \mathtt{mov}(v_1, v_2), \Gamma, \Pi\rangle \xrightarrow{\gamma} \mathtt{sh\_mov}(v_1, v_2)$$

$$\Gamma \vdash v_1 = \mathit{tainted} \quad \Rightarrow \quad \langle \mathtt{add}(v_1, v_2, v_2), \Gamma, \Pi\rangle \xrightarrow{\gamma} \mathtt{sh\_mov}(v_1, v_2, v_3)$$

$$\exists x \in \Pi[v_0], \Gamma \vdash x = \mathit{tainted} \quad \Rightarrow \quad \langle \mathtt{stm}(v_0, v_1), \Gamma, \Pi\rangle \xrightarrow{\gamma} \mathtt{sh\_stm}(v_0, v_1)$$

$$\Gamma \vdash v_1 = \mathit{tainted} \quad \Rightarrow \quad \langle \mathtt{ldm}(v_1, v_0), \Gamma, \Pi\rangle \xrightarrow{\gamma} \mathtt{sh\_ldm}(v_1, v_0)$$

Figure 7. The relation γ that partially instruments Angels programs.



```
0: sh_adr(e0, f)

1: sh_phi(e1, e0, e2)
   phi(a1, a0, a2)
   bzr(a1, 9)

9: sh_out(e1)
   end

4: add(a2, a1, b)
   ldm(c, e1)
   out(c)
   sh_add(e2, e1, d)
   jmp(l)
```

Figure 8. (a) The result of the static analysis applied on the program seen in Figure 4(a). (b) Partially instrumented program.

3. Experiments

We have implemented our algorithm on top of the LLVM compiler [Lattner and Adve 2004], and have tested it in an Intel Core 2 Duo Processor with a 2.20GHz clock, and 2 GB of main memory on a 667 MHz DDR2 bus. The operating system is Ubuntu 11.04.

The benchmarks: We have run our instrumentation framework on programs taken from two public benchmarks. The first, `FreeBench` (FB) [1], is used to measure system's workloads. This benchmark consists of six applications that include cryptography algorithms, neural networks and data compression. The second benchmark suite, shootout (ST) [2], consists of eight programs used in a popular website that compares different programming languages. Some characteristics of these benchmarks are given by Figure 9. In this section we will consider that an address leaks to the outside world if it is used as an ar-

[1]`http://code.google.com/p/freebench/`

[2]`http://shootout.alioth.debian.org/`

Benchmark	Application	LoC	Sinks	Sources	Assembly
ST	`fasta`	133	0	23	218
ST	`fannkuch`	104	3	27	281
ST	`n-body`	141	9	49	463
ST	`nsieve-bits`	35	5	7	130
ST	`partialsums`	67	1	38	212
ST	`puzzle`	66	1	12	210
ST	`recursive`	54	1	6	174
ST	`spectral-norm`	52	2	20	258
FB	`distray`	447	8	260	1,379
FB	`fourinarow`	702	1	109	2,124
FB	`mason`	385	11	303	690
FB	`neural`	785	16	156	1,425
FB	`pcompress2`	903	14	180	1,606
FB	`pifft`	4,185	3	1,857	19,320

Figure 9. Characteristics of the benchmarks. LoC: number of lines of C. Assembly: number of instructions in the binary representation of the original program.

gument of the standard `printf(char*, ...)` function. Thus, "sinks", in Figure 9 is the total number of occurrences of the `printf` function in the text of the application. Sources are instructions that produce address information, such as the ampersand (&) operator.

The results of the static analysis: Figure 10 shows the results of our static analysis. We compare the size of the constraint graph that the static analysis builds for each program with the fraction of that graph that has been marked as tainted. The tainted nodes belong into a path from a source function, e.g., code that generates address information, to a sink function, e.g., code that disclosures address information. As it is possible to see, about one fourth of each constraint graph is marked as tainted. In other words, the program slice that must be instrumented corresponds to about one fourth of the original program.

Instrumentation overhead: Figure 11 compares the runtime overhead of full and partial instrumentation. We have fed each program – original, fully instrumented and partially instrumented – with the reference input of each benchmark. As we can see, the full instrumentation imposes a non-trivial overhead on the programs. On average, fully instrumented programs are 4.9x, e.g., $(74.77secs/15.25secs)$ slower than the original programs. The partially instrumented programs are only 1.4x, e.g., $(22.04secs/15.25secs)$ slower. However, there are some cases, such as `pifft` and `puzzles`, in which the partially instrumented programs are considerably slower. These large slowdowns happen because the instrumentation code is inserted inside critical paths in the program code, such as deeply nested loops.

4. Related Work

We believe that this paper describes the first attempt to detect dynamically address disclosure vulnerabilities. However, much work has been done to detect dynamically other types of vulnerabilities. For instance, in 2005, James Newsome designed and imple-

Application	CG	TG	%Tainted
`fasta`	91	23	25.27%
`fannkuch`	118	27	22.88%
`n-body`	171	49	28.65%
`nsieve-bits`	70	7	10.00%
`partialsums`	192	38	19.79%
`puzzle`	80	25	31.25%
`recursive`	91	6	6.59%
`spectral-norm`	121	20	16.53%
`distray`	993	260	26.18%
`fourinarow`	1242	109	8.78%
`mason`	704	303	43.04%
`neural`	855	156	18.25%
`pcompress2`	929	180	19.38%
`pifft`	7669	1901	24.79%

Figure 10. Data produced by the static analysis. CG: number of nodes in the program's constraint graph. TG: number of tainted nodes in the constraint graph. %Tainted = TG/CG.

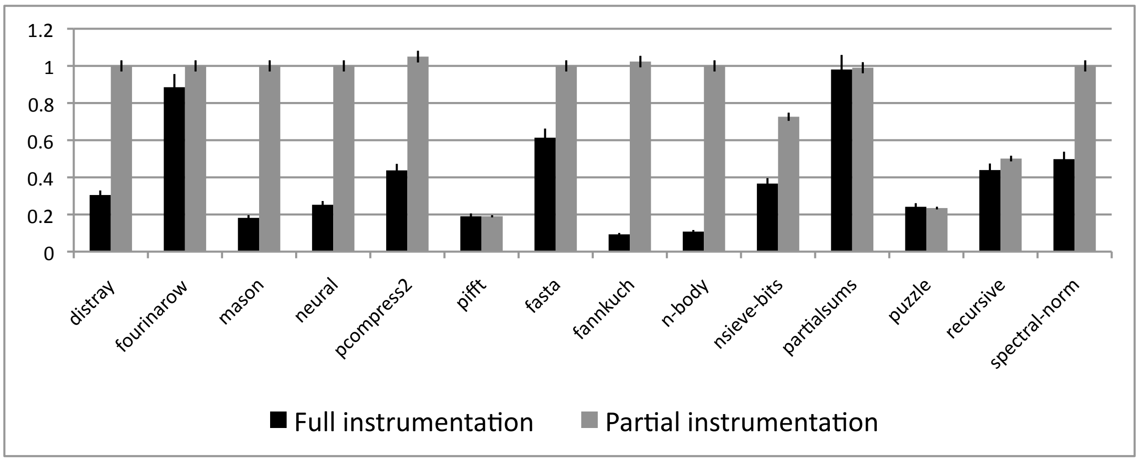


Figure 11. Runtime of the instrumented program (full or partial) divided by the runtime of the original program. The shorter the bar, the slower is the program.

mented a dynamic instrumentation framework that, contrary to previous approaches, was able to generate very efficient code [Newsome and Song 2005]. Newsome has been able to achieve this efficiency by applying different optimizations on the instrumented code. One of these optimizations, *fast path*, consists in duplicating the program code, in such a way that non-instrumented code is executed if the program inputs are not tainted. Similarly, Clause *et al.* [Clause et al. 2007] have proposed an instrumentation library that can be customized to track different forms of data. Clause *et al.*'s tool is not as efficient as the framework proposed by Newsome; however, it is much more general. None of these previous work has tracked address leaks. They also did not use static analysis to mitigate the instrumentation overhead.

There are many previous works that have proposed to combine static and dynamic analysis to secure programs. Huang *et al.*, for instance, have used static analysis to reduce the amount of runtime checks necessary to guard PHP programs against SQL injection attacks [Huang et al. 2004]. Similar approaches have been proposed by Zhang *et al.* [Zhang et al. 2011], Balzarotti *et al.* [Balzarotti et al. 2008] and Keromytis *et al.* [Keromytis et al. 2011]. None of these previous works tracks address leaks.

Finally, there is a large body of literature related to the use of pure static analysis to detect information flow vulnerabilities [Rimsa et al. 2011, Wassermann and Su 2007, Xie and Aiken 2006]. These works deal mostly with information flow in dynamically typed languages, such as PHP. The main problem of concern is to detect tainted flow vulnerabilities. We say that a program contains a tainted flow vulnerability if an adversary can feed it with malicious data, and this data reaches a sensitive function. Contrary to us, none of these previous works deal with address leaks, neither try to instrument the program to secure it dynamically.

5. Conclusion

This paper has presented a framework that tracks information leaks at runtime. This framework instruments the source code of the target program, shadowing every program variable and memory location that it uses. These shadow values might be in one of two states: clean or tainted. If the instrumented program tries to output a value shadowed as tainted, then we fire an exception, and the program terminates, providing a log back to the end user. This type of source code instrumentation imposes a heavy burden on the target program. In order to avoid this overhead, we rely on a static analysis that proves that parts of the program do not need to be instrumented. This static analysis has been modeled as the combination of two type inference engines: the first propagates tainted information forwardly, the second propagates information backwards. We have show empirically that this static analysis is effective, being able to reduce the instrumentation overhead from 5 to 1.44 times.

Future works: Although we are able to correctly instrument large programs, our framework is still a research artifact. For instance, some partially instrumented programs are still much slower than the original programs. Our next goal is to decrease this overhead. We want to apply program optimizations on the instrumented code, such as Qin's style code coalescing [Qin et al. 2006], a technique that has been developed to improve the runtime of binary instrumentation frameworks.

Reproducibility: All the benchmarks used in Section 3 are publicly available at their websites. The LLVM compiler is open source. Our code and further material about this project, including a Prolog interpreter of Angels, is available at `http://code.google.com/p/addr-leaks/`.

References

Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. (2008). Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *SP*, pages 387–401. IEEE Computer Society.

Bhatkar, E., Duvarney, D. C., and Sekar, R. (2003). Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *USENIX Security*, pages 105–120.

Buchanan, E., Roemer, R., Shacham, H., and Savage, S. (2008). When good instructions go bad: generalizing return-oriented programming to RISC. In *CCS*, pages 27–38. ACM.

Clause, J., Li, W., and Orso, A. (2007). Dytan: a generic dynamic taint analysis framework. In *ISSTA*, pages 196–206. ACM.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490.

Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D.-T., and Kuo, S.-Y. (2004). Securing web application code by static analysis and runtime protection. In *WWW*, pages 40–52. ACM.

Keromytis, A. D., Stolfo, S. J., Yang, J., Stavrou, A., Ghosh, A., Engler, D., Dacier, M., Elder, M., and Kienzle, D. (2011). The minestrone architecture combining static and dynamic analysis techniques for software security. In *SYSSEC*, pages 53–56. IEEE Computer Society.

Lattner, C. and Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE.

Levy, E. (1996). Smashing the stack for fun and profit. *Phrack*, 7(49).

Newsome, J. and Song, D. X. (2005). Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*. USENIX.

Qin, F., Wang, C., Li, Z., Kim, H.-s., Zhou, Y., and Wu, Y. (2006). LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO*, pages 135–148. IEEE.

Quadros, G. S. and Pereira, F. M. Q. (2011). Static detection of address leaks. In *SBSeg*, pages 23–37.

Rimsa, A. A., D'Amorim, M., and Pereira, F. M. Q. (2011). Tainted flow analysis on e-SSA-form programs. In *CC*, pages 124–143. Springer.

Schwartz, E. J., Avgerinos, T., and Brumley, D. (2010). All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *S&P*, pages 1–15. IEEE.

Shacham, H. (2007). The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *CCS*, pages 552–561. ACM.

Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. (2004). On the effectiveness of address-space randomization. In *CSS*, pages 298–307. ACM.

Wassermann, G. and Su, Z. (2007). Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI*, pages 32–41. ACM.

Xie, Y. and Aiken, A. (2006). Static detection of security vulnerabilities in scripting languages. In *USENIX-SS*. USENIX Association.

Zhang, R., Huang, S., Qi, Z., and Guan, H. (2011). Combining static and dynamic analysis to discover software vulnerabilities. In *IMIS*, pages 175–181. IEEE Computer Society.