The Address Disclosure Vulnerability

Author names removed due to blind review
Institution name removed due to blind review

***Abstract*—An address disclosure vulnerability 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 uses a dynamic monitor to prevent address leaks 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 have tested our implementation on a benchmark suite with 434 programs, that includes SPEC CPU 2006, and gives us approximately 2 million lines of C. We have been able to effectively exploit 11, out of 19 warnings produced by our flow analysis. The combination of static and dynamic analyses provide us with a practical way to secure software against address disclosures. A fully instrumented program is, on average, almost 500% slower than the original benchmark. However, the static analysis has been able to reduce this overhead to less than 3% on average.**

***Keywords*-Address disclosure; Information flow analysis; Semantics; Type system; Dynamic analysis; Static analysis**

I. Introduction

Modern operating systems use a protection mechanism called *Address Space Layout Randomization* (ASLR) [3], [29]. 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* [29] and *return-oriented-programming* (ROP) [5], [28]. 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.* [29] 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 [3, 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 uncover and prevent in this paper.

The first contribution of this paper is the description of the address disclosure vulnerability itself. Although this problem has been continuously discussed in mailing lists and blogs by software security experts and enthusiasts, so far it has not been approached under an academic perspective. We describe address leaks informally in Section II, and show an actual example of code exploit that relies on address disclosure to circumvent protection mechanisms used by modern operating systems. We then formalize our presentation in Section III, via a core programming language that not only defines address disclosures, but also grounds the techniques that we use to deal with this problem.

In Section III we also show how to combine static and dynamic analyses to secure programs against address disclosures. The techniques that we use in this paper, instrumentation and information flow analysis, are not new; however, this is the first paper to use them to prevent address leaks. Furthermore, some aspects of our flow analysis, like its division into a forward and a backward components, and the heavy use of points-to analysis, seem to be unique to this work. In the dynamic side, we have designed an instrumentation framework that automatically converts a program into a software that cannot contain address leaks. This transformation consists in instrumenting every program operation that propagates data, be it in registers or in memory. In this way, we know which information might give address knowledge to an adversary, and which information might not. If harmful information reaches an output point that the 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 attacker to perform exploits that require address information to succeed.

Non-surprisingly, the instrumentation imposes on the target program a heavy runtime overhead: the sanitized program can be as much as 10x slower than the original code. To mitigate this overhead, we have developed an information flow analysis that proves that parts of the code do not need to be instrumented. As we show in Section III-A, this static analysis points, conservatively, which instructions are data dependent on address information. The innocuous operations that the static analysis identifies are not instrumented. Our static analysis incorporates state-of-the-art features to achieve good precision: it is field and flow sensitive, it is interprocedural, it is object sensitive, i.e., it distinguishes one object from another of the same type [14], it models the memory heap via pointer analysis, and it achieves a limited form of context sensitiveness via function inlining.

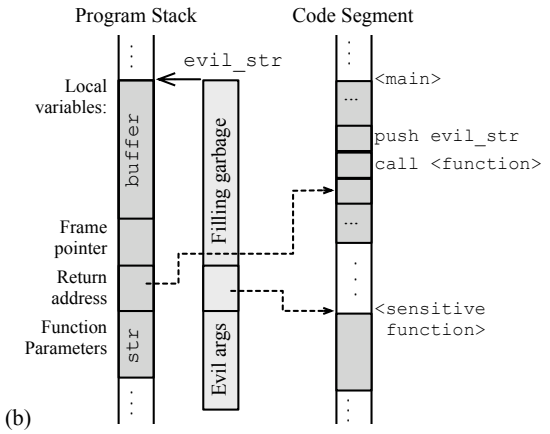
We have implemented our instrumentation framework, and the companion static analysis in the LLVM compiler [18]. Thus, contrary to binary instrumentation tools [20], [36], presently we only instrument programs if we have access to their source code. We chose to stay at the intermediate representation level because this decision makes it easier to combine our instrumentation library with the static analysis. The price that we have to pay for our choice comes in the form of false-positive warnings that the instrumented programs report dynamically: information that comes out of unknown libraries is always marked as dangerous. We can reduce the false-positives by adding functions known to be harmless to a white list.

As a proof of concept, we chose to search for address information that escapes through `libc`'s `printf` function. Other sensitive functions can be easily added to our framework. As we show in Section IV, we have used our tool to analyze a test suite that contains almost 2 million lines of code, and that includes SPEC CPU 2006. Instrumented programs can be 2% to 1,070% slower than the original programs, with an average slowdown of 76.88%. 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 22%, with an average of 1.71%. This overhead is low enough to justify the use of our dynamic monitor in production code. Our static analysis has reported 19 warnings in our test suite. Manual inspection of these warnings has shown that 11 of them were true address leaks.

```
void function(char* str) {
   char buffer[16];
   strcpy(buffer,str);
}

void main() {
  ...
  char* evil_str = read_data();
  function(evil_str);
  ...
}
```

(a)



(b)

Figure 1. (a) A program that contains a buffer overflow vulnerability. (b) An schematic example of a stack overflow. The return address of `function` is diverted by a maliciously crafted input to another procedure.

II. Background

A *buffer*, also called an array or vector, is a contiguous sequence of elements stored in memory. Some programming languages, such as Java, Python and JavaScript are *strongly typed*, which means that they only allow combinations of operations and operands that preserve the type declaration of these operands. As an example, all these languages provide arrays as built-in data structures, and they verify if indexes are within the declared bounds of these arrays. There are other languages, such as C or C++, which are *weakly typed*. They allow the use of variables in ways not predicted by the original type declaration of these variables. C or C++ do not check array bounds, for instance. Thus, one can declare an array with n cells in any of these languages, and then read the cell at position $n + 1$. This decision, motivated by efficiency [30], is the reason behind an uncountable number of worms and viruses that spread on the Internet [3].

Programming languages normally use three types of memory allocation regions: static, heap and stack. Global variables, runtime constants, and any other data known at compile time usually stays in the static allocation area. Data structures created at runtime, that outlive the lifespan of the functions where they were created are placed on the heap. The activation records of functions, which contain, for instance, parameters, local variables and return address, are allocated on the stack. In particular, once a function is called, its return address is written in a specific position of its activation record. After the function returns, the program resumes its execution from this return address.

A *buffer overflow* consists in writing in an array a quantity of data large enough to go past its upper bound; hence, overwriting other program or user data. It can happen in the stack or in the heap. In the *stack overflow* scenario, by carefully crafting this input string, one can overwrite the return address in a function's activation record; thus, diverting execution to another code area. Earlier buffer overflow attacks included the code that should be executed in the input array [19]. However, modern operating systems mark writable memory addresses as non-executable – a protection known as *Write⊕Execute* [29, p.299], or Data Execution Prevention (DEP). Therefore, attackers tend to divert execution to system functions such as `chmod` or `sh`, if possible. Usually the malicious string contains also the arguments that the cracker wants to pass to the sensitive function. Figure 1 illustrates an example of buffer overflow.

A buffer overflow vulnerability gives crackers control over the compromised program even when the operating system does not allow function calls outside the memory segments

allocated to that program. Attackers can call functions from `libc`, for instance. This library, which is share-loaded in every UNIX system, allows users to fork processes and to send packets over a network, among other things. Such attack is called *return to* `libc` [29]. Return to `libc` has been further generalized to a type of attack called *return-oriented-programming* (ROP) [28]. If a binary program is large enough, then it is likely to contain many bit sequences that encode valid instructions. Hovav Shacham [28] has shown how to derive a Turing complete language from these sequences in a CISC machine, and Buchanan *et al.* [5] have generalized this method to RISC machines.

There exist ways to prevent these types of "return-to-known-code" attacks. The best known defense mechanism is *address obfuscation* [3]. A compiler can randomize the location of functions inside the binary program, or the operating system can randomize the virtual address of shared libraries. Shacham *et al.* [29] have shown that these methods are susceptible to brute force attacks; nevertheless, address obfuscation slows down the propagation rate of worms that rely on memory corruption vulnerabilities substantially. Address obfuscation is not, however, the ultimate defense mechanism. If the target program leaks some of its internal addresses through a public channel, then attackers can calculate the addresses of sensitive operations that they want to call. In the rest of this section we show, via an example, how to exploit a system protected by ASLR and DEP.

A. Address Leak in one Example

We illustrate the address disclosure vulnerability via the echo server in Figure 2. 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 [19], because it forbids the execution of writable memory space. However, we can use a buffer overflow to divert program execution to one of the system's functions. In this example we shall open a telnet terminal in the server's machine. This type of attack depends on the adversary knowing the address of a sensitive function, e.g., `libc`'s `system` in this example. This information is not easily available in an ASLR protected system, unless the target software contains an address leak.

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 Figure 3. 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:

```
1 void *libc;
2 void process_input(char *inbuf, int len, int clientfd) {
3     char localbuf[40];
4     if (!strcmp(inbuf, "debug\n")) {
5         sprintf(localbuf, "localbuf %p\nsend() %p\n",
                  localbuf, dlsym(libc, "send"));
6     } else { memcpy(localbuf, inbuf, len); }
7     send(clientfd, localbuf, strlen(localbuf), 0);
8 }
9 int main() {
10    int sockfd, clientfd, c_len, len;
11    char inbuf[5001];
12    struct sockaddr_in myaddr, addr;
13    libc = dlopen("libc.so", RTLD_LAZY);
14    sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
15    myaddr.sin_family = AF_INET;
16    myaddr.sin_addr.s_addr = htonl(INADDR_ANY);
17    myaddr.sin_port = htons(4000);
18    bind(sockfd, (struct sockaddr *)&myaddr, sizeof(myaddr));
19    listen(sockfd, 5);
20    c_len = sizeof(addr);
21    while (1) {
22        clientfd = accept(sockfd, (struct sockaddr *)&addr,
                  &c_len);
23        len = recv(clientfd, inbuf, 5000, 0);
24        inbuf[len] = '\0';
25        process_input(inbuf, len + 1, clientfd);
26        close(clientfd);
27    }
28    close(sockfd); dlclose(libc);
29    return 0;
30 }
```

Figure 2. An echo server implemented in C.

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

Figure 3. A Python program that explores the address disclosure in the program from Figure 2.

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.

III. 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 4. 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 5 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 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 any function that can be regarded as sensitive. 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 [9]. 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 4 shows the syntax of ϕ-functions. This representation is not necessary for computational completeness; however, as we will see in Section III-A, 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.

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

Figure 4. The syntax of Angels.

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

Figure 5. Examples of syntax of C mapped to instructions of Angels.

Figure 6 describes the small-steps operational semantics of Angels. Our formalization is similar to Schwartz *et al.*'s [27, Fig.5], yet much lower level. 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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 6. The Operational Semantics of Angels.

us the path used to reach the ϕ-function.

The memory Σ 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 variables to their integer addresses in Σ. We represent the output channel Θ as a list. 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 7 shows an example of a program written in Angels. The program in Figure 7(a) prints the contents of an array, and then the base address of the array itself. Figure 7(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 7 contains an address disclosure vulnerability, a notion that we introduce in Definition 3.1. 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 3.1: THE ADDRESS DISCLOSURE VULNERABILITY: "Let P be an Angels program, such that $\langle P, 0, 0, \lambda x.0, []\rangle \to \langle \Sigma, \Theta\rangle$, where $\lambda x.0$ is the environment that maps variables to zero, and [] is the empty output channel. P contains an address disclosure vulnerability if an adversary can discover $\Delta(v)$ for some v used in P, given the knowledge of Θ plus the source code of P."

The Instrumentation language. In order to secure a program against address disclosures, we will instrument it.

(a)

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

(b)

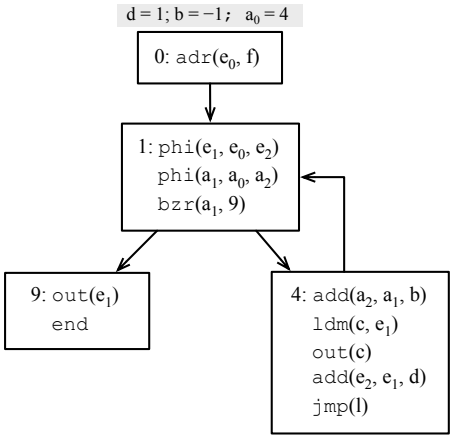


Figure 7. A C program translated to Angels.

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 instrumentation 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 8. 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 6, augmented with the six new rules given in Figure 8. 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 9 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*.

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

Figure 9. The full instrumentation of an angels program.

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.

Correctness of Full Instrumentation: The instrumentation, i.e., the ι relation from Figure 9, should not modify the semantics of the original program. We prove this property in Theorem 3.2. Moreover, the ι relation should ensure that an instrumented program does not contain an address leak. Theorem 3.5 proves this property. Both theorems require the notion of an *execution trace*. The execution trace of a program P is the sequence T of instructions that were processed during the execution of P. We will use the following notation: If T is the execution trace of $\langle P, 0, 0, \lambda x.0, [] \rangle$, and $P \xrightarrow{\iota} P^{\iota}$, then T^{ι} is the execution trace of $\langle P^{\iota}, 0, 0, \lambda x.0, [] \rangle$. In the rest of this section, we rely on two assumptions: (i) every variable is initialized before being used, and (ii) there exist no overlap between original and shadow memory.

Theorem 3.2: If variable v is assigned a value x at the i-th instruction of T^{ι}, then variable v is assigned the same value x at the i-th instruction of T.

Proof: The proof is by induction on the size of T^{ι}. Assume the theorem holds for the first N instructions of T^{ι}. Perform case analysis on each rule in Figure 6, and in the corresponding rule in Figure 8:

- ADRSEM/INS: v_1 is assigned $\Sigma[\Delta(v_2)]$ in both cases.
- OUTSEM/INS: the `out` instruction does not define variables.
- MOVSEM/INS: $\Sigma[\Delta(v_1)]$ is assigned $\Sigma[\Delta(v_2)]$ in both cases. By induction, $\Sigma[\Delta(v_2)]$ is the same in both programs.
- STMSEM/INS: by induction, $\Sigma[\Delta(v_0)]$ and $\Sigma[\Delta(v_1)]$ are the same in both programs. Hence, $\Sigma[x \mapsto n]$ in both of them. By hypothesis, $\Sigma[\Delta(v_1) + D]$ has not being modified by assignments in the original program, and an assignment to $\Sigma[x + D]$ does not modify any memory in the original program.

The other rules are similar. □

As a direct corollary of Theorem 3.2, we have that any execution trace of the instrumented program is a prefix of

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

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

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

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

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

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

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

Figure 8. The Operational Semantics of the instrumented instructions.

the execution trace in the original program:

Corollary 3.3: T^ι is a prefix of T.

Proof: the two rules that determine control flow, BzrSem and BnzSem are the same for the original and the instrumented programs. By Theorem 3.2, branches always happen on the same values in both programs. □

Lemma 3.4 states that tainted information propagates in the shadow memory according to the data dependences in the original program. We say that a variable v is *data dependent* on a variable u, in a program P, if (i) v is defined by an instruction that uses u, i.e., $\texttt{mov}(v, u) \in P$; or, (ii) $\Delta(v_0) = v$, and $\texttt{stm}(v_0, u) \in P$; or, (iii) $\Delta(v_0) = u$, and $\texttt{ldm}(v, v_0) \in P$; or (iv) v is data dependent on some x, and x is data dependent on u.

Lemma 3.4: Let P be an angels program, and P^ι be such that $P \xrightarrow{\iota} P^\iota$. Variable v is data dependent on some address at the i-th instruction of T^ι, if, and only if, $\Sigma[\Delta(v) + D]$ contains the value *tainted*.

Proof: We will prove the "if" part of the theorem. The proof for the "only-if" side of it is available in our technical report. We proceed by induction on the size of the execution trace T^ι, assuming that the hypothesis is true for every prefix T^ι_p of T^ι, such that $|T^\iota_p| < N$. We perform a case analysis on the instruction $I^\iota \in T^\iota$ that comes after T^ι_p:

- $\texttt{sh\_adr}(v_1, v_2)$: the only variable modified is v_1. By Rule AdrIns, $\Sigma[\Delta(v_1) + D] \mapsto \mathit{tainted}$.
- $\texttt{sh\_out}(v)$: no variable is modified.
- $\texttt{sh\_mov}(v_1, v_2)$: the hypothesis holds for v_2, by induction. By definition, v_1 is data dependent on v_2. The taint state propagates from v_2 to v_1 by MovIns.
- $\texttt{sh\_stm}(v_0, v_1)$: if $\Sigma[\Delta(v_0)] = x$, then the hypothesis holds for x by induction. By definition, v_0 is data dependent on v_1. By Rule StmIns, the same taint state propagates from x to v_0.

The other rules are similar. □

Theorem 3.5: 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 output values that are data-dependent on address information.

Proof: According to the premises of Rule OutIns, information is printed in the output channel only if it is not associated to a tainted state. By Lemma 3.4 only information that is data dependent on address information is associated with tainted states. □

A. Reducing Instrumentation Overhead via Static Analysis.

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 describe this static analysis as a type system, which we solve via the combination of a forward and a backward propagation engine.

Points-to Analysis. The type system that we propose is parameterized by a points-to analysis. A solution to points-to analysis in Angels is a map Π of variables to points-to facts, which we obtain as the fixed point of the constraint system given in Figure 10. A points-to fact is a set of program variables, in such a way that if variable v might hold the

Instruction	Constraint
$\texttt{adr}(v_1, v_2)$	$\{v_2\} \subseteq \Pi(v_1)$
$\texttt{mov}(v, v_1)$	$\Pi(v_1) \subseteq \Pi(v)$
$\texttt{stm}(v_0, v_1)$	$x \in \Pi(v_0) \Rightarrow \Pi(v_1) \subseteq \Pi(x)$
$\texttt{ldm}(v_1, v_0)$	$x \in \Pi(v_0) \Rightarrow \Pi(x) \subseteq \Pi(v_1)$

Figure 10. Constraints that define the points-to analysis.

address of variable x, then x is in the points-to set of v. We will not give any algorithm to solve the constraint system in Figure 10, because plenty of them have already been described in the literature. In our implementation, we use Hardekopf's *lazy cycle detection* [15]. This algorithm has an $O(|V|^3)$ worst-case asymptotic complexity, where $|V|$ is the number of variables in the input program. However, in practice the algorithm is much faster, due to a heuristics that finds cycles in graphs, as we show in Section IV, Fig. 21.

A type system to statically detect address leaks. We now describe a type system that annotates each program variable with one of three types: *clean*, *maybe* or *tainted*. The type system is described by a forward, and a backward components. The forward component first annotates variables with either the *clean* or the *maybe* types. The backward component then annotates the *maybe* variables with either the *clean* type, or the *tainted* type. Only variables that the backwards analysis annotates as *tainted* can leak address information. The forward inference engine is defined by the relation $\xrightarrow{\gamma\rightarrow}$, given in Figure 12. The backward inference engine is defined by the relation $\xrightarrow{\gamma\leftarrow}$, given in Figure 13. These analyses use the meet operators defined in Figure 11.

As we have mentioned in the beginning of this section, we work with programs in the SSA intermediate representation. The main advantage of using the SSA format is that the abstract state of a variable, e.g, *clean* or *tainted*, is invariant along every program point where the variable exists. Thus, we can bind this information directly to the variable, instead of to pairs formed by variables and program points, as more traditional analyses do. Thus, the SSA representation lets us perform our static analysis *sparsely*. As Choi *et al.* have demonstrated two decades ago [7], sparse analyses are more efficient in terms of speed and memory.

The forward analysis assumes that every variable is *clean*. Upon finding those variables that may hold addresses, it marks them with the *maybe* type. By Rule ADRFWD, we see that `adr` is the only instruction that can change the type of a variable during the forward analysis. The goal of the backward analysis, on the other hand, is to find which variables typed as *maybe* can reach the `out` instruction. From Rule OUTBKW, we see that `out` is the only instruction that can dump the *tainted* type into the typing environment.

Both the forward and the backward propagation algorithms have $O(|V|^2)$ implementations. Lets look into the

$\sqcap_{\rightarrow}$	$clean$	$maybe$
$clean$	$clean$	$maybe$
$maybe$	$maybe$	$maybe$

$\sqcap_{\leftarrow}$	$maybe$	$tainted$	$clean$
$maybe$	$maybe$	$tainted$	$clean$
$tainted$	$tainted$	$tainted$	$clean$
$clean$	$clean$	$clean$	$clean$

Figure 11. Abstract semantics of the meet operators used in our forward and backward type inference engine.

$$[\text{RF}] \quad \langle \texttt{adr}(v_1, v_2), \Gamma, \Pi\rangle \xrightarrow{\gamma\rightarrow} \Gamma[v_1 \mapsto \mathit{maybe}]$$

$$[\text{MF}] \quad \frac{\Gamma \vdash v_1 = t_1 \qquad \Gamma \vdash v_2 = t_2 \qquad t_1' = t_1 \sqcap_{\rightarrow} t_2}{\langle \texttt{mov}(v_1, v_2), \Gamma, \Pi\rangle \xrightarrow{\gamma\rightarrow} \Gamma[v \mapsto t_1']}$$

$$[\text{AF}] \quad \frac{\Gamma \vdash v_2 = t_2 \qquad \Gamma \vdash v_3 = t_3 \qquad t_1 = t_2 \sqcap_{\rightarrow} t_3}{\langle \texttt{add}(v_1, v_2, v_3), \Gamma, \Pi\rangle \xrightarrow{\gamma\rightarrow} \Gamma[v_1 \mapsto t_1]}$$

$$[\text{PF}] \quad \frac{\Gamma \vdash v_1 = t_1 \ldots \Gamma \vdash v_k = t_k \qquad t = t_1 \sqcap_{\rightarrow} \ldots \sqcap_{\rightarrow} t_k}{\langle \texttt{phi}(v, \{v_1, \ldots, v_k\}), \Gamma, \Pi\rangle \xrightarrow{\gamma\rightarrow} \Gamma[v \mapsto t]}$$

$$[\text{SF}] \quad \frac{\Gamma \vdash v_1 = t_1 \qquad \Gamma \vdash x = t_x \qquad t_x' = t_1 \sqcap_{\rightarrow} t_x}{\langle \texttt{stm}(v_0, v_1), \Gamma, \Pi\rangle \xrightarrow{\gamma\rightarrow} \Gamma[x \mapsto t_x]}, x \in \Pi[v_0]$$

$$[\text{LF}] \quad \frac{\Gamma \vdash v_1 = t_1 \qquad \Gamma \vdash x = t_x \qquad t_1' = t_1 \sqcap_{\rightarrow} t_x}{\langle \texttt{ldm}(v_0, v_1), \Gamma, \Pi\rangle \xrightarrow{\gamma\rightarrow} \Gamma[v_1 \mapsto t_1']}, x \in \Pi[v_0]$$

Figure 12. The forward type inference analysis.

complexity of the forward analysis first. Before running it, we let $\Gamma[v] = \mathit{clean}$ for every variable v. The type of a variable can change only once, becoming *maybe*; thus, the total number of times we must analyze a variable is upper bounded by $O(1)$. We reach the quadratic complexity due to the use of points-to facts, which causes Rules SF and LF to have $O(|V|)$ complexity. We obtain the $O(|V|^2)$ complexity of the backward analysis in a similar way: the type of each variable can change only once, from *maybe* to *tainted*, and the points-to facts contribute another linear component to the processing of loads and stores.

Partial Instrumentation. Given the results of our type system, we can reduce the amount of instrumented code in the target program that the ι relation from Figure 9 inserts. With such purpose, we define a new instrumentation framework as the γ relation, which is given in Figure 14. Contrary to the algorithm in Figure 9, 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 variable with the *tainted* type. Theorem 3.8 proves that the partial instrumentation is correct. For the sake of space,

$$[\text{OB}] \quad \frac{\Gamma \vdash v = t \qquad t' = \mathit{tainted} \sqcap_{\leftarrow} t}{\langle \texttt{out}(v), \Gamma, \Pi \rangle \xrightarrow{\gamma_{\leftarrow}} \Gamma[v \mapsto t']}$$

$$[\text{MB}] \quad \frac{\Gamma \vdash v_1 = t_1 \qquad \Gamma \vdash v_2 = t_2 \qquad t_2' = t_1 \sqcap_{\leftarrow} t_2}{\langle \texttt{mov}(v_1, v_2), \Gamma, \Pi \rangle \xrightarrow{\gamma_{\leftarrow}} \Gamma[v_2 \mapsto t_2']}$$

$$[\text{AB}] \quad \frac{\begin{array}{c}\Gamma \vdash v_2 = t_2 \\ \Gamma \vdash v_3 = t_3 \qquad t_2' = t_1 \sqcap_{\leftarrow} t_2 \qquad t_3' = t_1 \sqcap_{\leftarrow} t_3\end{array}}{\langle \texttt{add}(v_1, v_2), \Gamma, \Pi \rangle \xrightarrow{\gamma_{\leftarrow}} (\Gamma[v_2 \mapsto t_2])[v_3 \mapsto t_3']}$$

$$[\text{PB}] \quad \frac{\Gamma \vdash v = t \qquad \Gamma \vdash v_i = t_i \qquad t_i' = t \sqcap_{\leftarrow} t_i}{\langle \texttt{phi}(v, \{\ldots, v_i, \ldots\}), \Gamma, \Pi \rangle \xrightarrow{\gamma_{\leftarrow}} \Gamma[v_i \mapsto t_i']}$$

$$[\text{SB}] \quad \frac{\Gamma \vdash x = t_x \qquad \Gamma \vdash v_1 = t_1 \qquad t_1' = t_x \sqcap_{\leftarrow} t_1}{\langle \texttt{stm}(v_0, v_1), \Gamma, \Pi \rangle \xrightarrow{\gamma_{\leftarrow}} \Gamma[v_1 \mapsto t_1']}, x \in \Pi[v_0]$$

$$[\text{LB}] \quad \frac{\Gamma \vdash x = t_x \qquad \Gamma \vdash v_1 = t_1 \qquad t_x' = t_x \sqcap_{\leftarrow} t_1}{\langle \texttt{ldm}(v_1, v_0), \Gamma, \Pi \rangle \xrightarrow{\gamma_{\leftarrow}} \Gamma[x \mapsto t_x']}, x \in \Pi[v_0]$$

Figure 13. The backward type inference analysis.

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

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

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

$$\frac{\Gamma \vdash v_1 = \mathit{tainted}}{\langle \texttt{add}(v_1, v_2, v_2), \Gamma, \Pi \rangle \xrightarrow{\gamma} \texttt{sh\_add}(v_1, v_2, v_3)}$$

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

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

Figure 14. The relation γ that partially instruments Angels programs.

we omit from Figure 14 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.

Figure 15 shows the result of applying the static analysis onto the program seen in Figure 7(b). Figure 15(a) shows the *dependence graph* [12] that the target program induces. Each variable in this graph has been annotated with the results of the forward inference engine. Thus, a variable may have been assigned the type *clean* (C) or *maybe* (M). The final type assignment, as found by the backward annotation is shown in Figure 15(b). 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, as we see in Figure 15(c).

Correctness of Partial Instrumentation: we now prove that partially instrumented programs do not contain address leaks. This proof consists in showing that any *dependence chain* between a source of address information and the output operation contains only instrumented instructions.

Lemma 3.6: Only variables that are data dependent on an address have the *maybe* type.

Proof: Only Rule RF binds the *maybe* type to a variable. A case analysis on each other rules shows that types only propagate from a variable v_1 to another variable v_2 if v_2 is data dependent on v_1. □

Lemma 3.7: A variable v has the *tainted* type if (i) it is data dependent on some address, and (ii) there exist $\texttt{out}(u)$ such that u is data dependent on v.

Proof: Only Rule OB binds the *tainted* type to a variable v. To prove (i), by the definition of $\mathit{tainted}$ (Fig. 11(bottom)), we have that $\mathit{tainted} \sqcap_{\leftarrow} t = \mathit{tainted}$ if $t = \mathit{maybe}$. By Lemma 3.6, v will have the *maybe* type if it is data dependent on some address. To prove (ii), a case analysis on each rule, but OB, shows that types only propagate from a variable v_1 to another variable v_2 if v_1 is data dependent on v_2. □

Theorem 3.8: 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 output values that are data-dependent on address information.

Proof: From Lemmas 3.6 and 3.7, if a variable is part of a dependence chain between an address information and an output instruction, than it has the *tainted* type. By the rules in Figure 14, we see that every instruction that defines a *tainted* variable is instrumented. Thus, any instruction that is in a dependence chain between address information and the output operation is instrumented. We conclude by applying Lemma 3.4 on the instrumented slice of the program. □

When Full Instrumentation does More than Static Analysis. The overhead imposed by the full instrumentation, according to the experiments that we show in Section IV, can be too high to production software. Nevertheless, the full instrumentation does not suffer from some limitations that exist in the static analysis. These limitations are not a shortcoming of our analysis in particular; on the contrary, they are present in any static analysis that assumes that the target program is well-defined. If the program that is analyzed contains non-deterministic behavior, then the static analysis might rely on wrong assumptions. Non-deterministic behavior may be present in C and C++ programs, due to the inherently unsafe type systems used in these languages. The program in Figure 16 illustrates this type of non-determinism.

The program in Figure 16 contains a buffer overflow vulnerability. If `N` is larger than the size of `buf2`, then

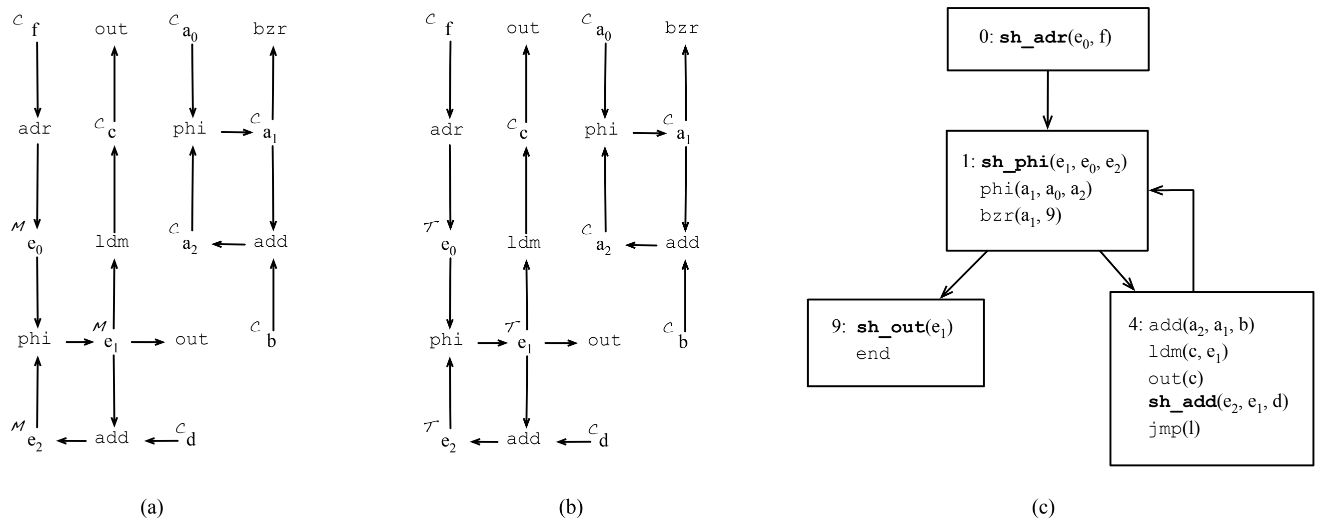


Figure 15. (a) Forward analysis applied on the program seen in Figure 7(a). (b) Backward analysis. (c) Partially instrumented program.

```
1 void fun1(char *s, int N) {
2     int *p = s;
3     char buf1[] = "hello";
4     char buf2[40];
5     strncpy(buf2, s, N);
6     printf("%s\n", buf1);
7 }
```

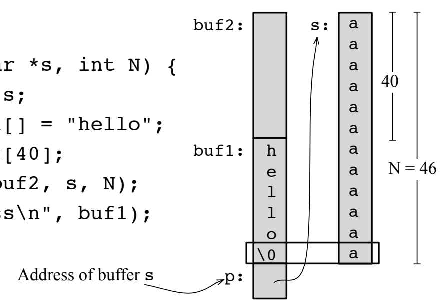


Figure 16. This C program may leak information. The static analysis does not catch this vulnerability, due to the badly defined semantics of out-of-bound array accesses in C.

```
void len(char* s) {
  unsigned w = 0;
  while(*s != '\0') {
    w++;
    s++;
  }
  printf("%u\n", w);
}
```

```
void print_adr() {
  int s = (int)&s;
  unsigned x = 0;
  unsigned m = ~(~0U>>1);
  while(m) {
    if (s & m)
      x |= m;
    m >>= 1;
  }
  printf("%u\n", x);
}
```

Figure 17. In function `len`, variable `w` is control dependent on variable `s`, which holds address information. The function `print_adr` contains an address leak, but our instrumentation does not catch it.

out-of-bounds memory writes will take place at line 5. An attacker can use this vulnerability to overwrite the null byte that marks the end of the string stored in `buf1`. If this overwrite happens, then the `printf` function at line 6 will reveal the contents of local variable `p`. In this example, `p` holds the address of the buffer `s`. The static analysis cannot detect this vulnerability, because it relies on a points-to analysis that assumes that the memory pointed by `buf1` and `buf2` is disjoint. However, the absence of dynamic bounds-access checks in C breaks this assumption. On the other hand, our full instrumentation library is able to detect this type of information leak.

Control Sensitiveness. The dynamic instrumentation rules shown in Figure 8 are not *control sensitive*. Those rules only consider data dependencies, and not control dependencies. For the formal semantics of a control sensitive dynamic monitor, we recommend the work of Russo and Sabelfeld [26, Figs.9-12]. We have implemented a control insensitive dynamic analysis to keep our approach practical. Had we done it otherwise, then we would have to deal with a high false-positive rate, because many loops are controlled by addresses, e.g., iterators in C++, or pointers in C. Figure 17 shows an example: variable `w`, in function `len`, is control dependent on the address information stored in variable `s`. However, an adversary cannot infer any pointer value from the output produced by function `len`.

We can handle control dependences by switching our representation from the SSA form to the Gated Static Single Assignment (GSSA) form [22]. However, we opted for not doing it in our production tool; thus, our implementation is unsound. Function `print_adr`, in Figure 17, illustrates an address disclosure vulnerability that we do not catch. The value in variable `s` – an address – is copied to variable `x`,

bit-by-bit. Because `x` is not data dependent on `s`, the rules in Figure 17 will not shadow it with the tainted state, and an unreported leak will happen at line 10. Nevertheless, whereas function `print_adr` is unlikely to exist in actual code, function `len` is common enough to justify our decision.

IV. Experiments

We have implemented all the algorithms described in this paper in the LLVM compiler [18]. In this section we evaluate our implementation empirically. The numbers that we show in this Section have been obtained in an AMD Athlon(tm) II P340 Dual-Core processor, with a 2.2GHz clock, 2GB of RAM, running Linux Ubuntu, version 12.04.1 LTS. We have tested our implementation on a suite of 434 programs, that include the entire LLVM test suite, plus the SPEC CPU 2006 benchmark collection. In total we have analyzed almost 2 million lines of C, which gave us 4.3 million instructions in the LLVM intermediate representation (bytecodes).

The Impact of Instrumentation on Code Size. Figure 18 shows the impact of our instrumentation library on the size of the 30 largest benchmarks in our suite. Non-surprisingly, the full instrumentation of a program tends to increases its size significantly, as we see in Figure 18 (bottom). If we consider our entire test suite, with 434 programs, then the code of the instrumented programs is 2.46 times larger than the code of the original programs. If we consider only the 30 benchmarks used in Figure 18 (bottom), then the code increases by 4.47 times.

Figure 18 (Top) shows how the static analysis reduces the impact of instrumentation on code size. Across the 434 benchmarks, we have observed an average code expansion of just 5.17%. Considering only the 30 largest benchmarks, which we show in Figure 18 (Top), we obtain a similar number: an increase of 5.06%. The partial instrumentation increases even the code of the programs that the flow analysis marks as completely secure, because of the library functions that are linked with the program to handle leaks. The main conclusion that we can derive from the two charts in Figure 18 is that the static analysis is essential to avoid code-size explosion when instrumenting programs to prevent address disclosures.

The Impact of Instrumentation on Runtime. Figure 19 compares the runtime of original and instrumented programs. Figure 19(bottom) shows the runtime impact of full instrumentation. Considering the 30 benchmarks seen in Figure 1, we observe a slowdown of 76.88% of the fully instrumented programs. As expected, the static analysis reduces this slowdown substantially. Figure 19(top) compares the runtime of the original and the partially instrumented programs. On average, the 30 partially instrumented programs are 1.71% slower than their unmodified versions.

Figure 20 shows static numbers that our flow analysis produces for the programs in SPEC CPU 2006. We are missing `400.perlbench`, because our distribution of LLVM does not compile it. The size of the dependence graph is roughly the number of bytecodes in the intermediate representation of the program, not counting branches and ϕ-nodes. The forward analysis, e.g., the rules in Figure 12, mark about 20-40% of the vertices in such graphs as possibly leaky. The backward analysis, on the other hand, has been able to mark every node in every dependence graph, except `bzip2`'s, as innocuous. This program - `bzip2` - has two paths from sources of address information to sinks.

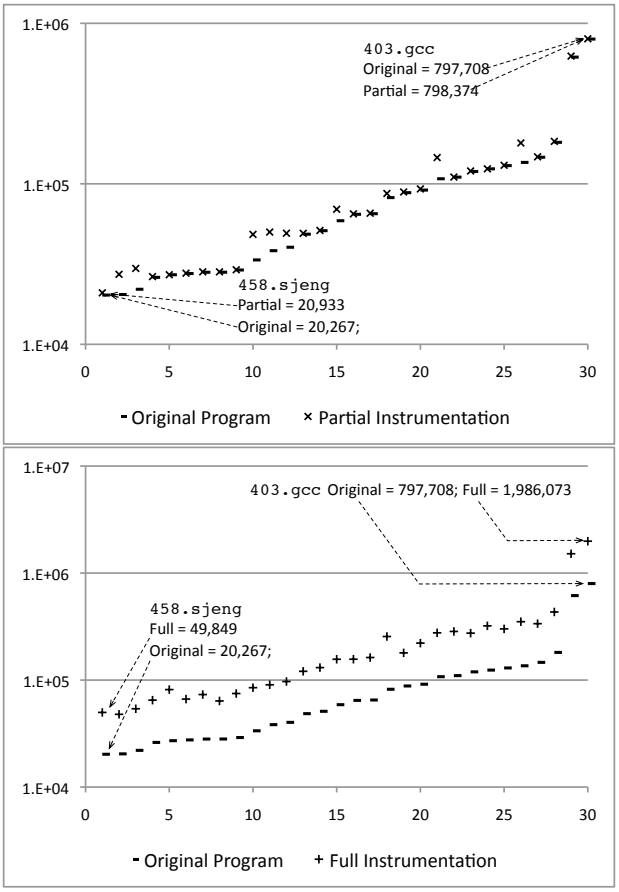


Figure 18. (Top) Size increase due to partial instrumentation. (Bottom) Size increase due to full instrumentation. Each dot corresponds to a benchmark. Sizes are given in number of LLVM bytecode instructions.

Runtime of the Static Analyses. Figure 21(Top) shows the time to run our tainted flow analysis and our implementation of points-to analysis for the 100 largest benchmarks in our test suite. The flow analysis is linear on the size of the program's dependence graph. Hence, it can be quadratic on the number of instructions in the program. However, this analysis is linear in practice, because these dependence graphs tend to be sparse. We have plotted the time to run the flow analysis together with the size of the programs, to provide a visual indication that our analysis is linear in

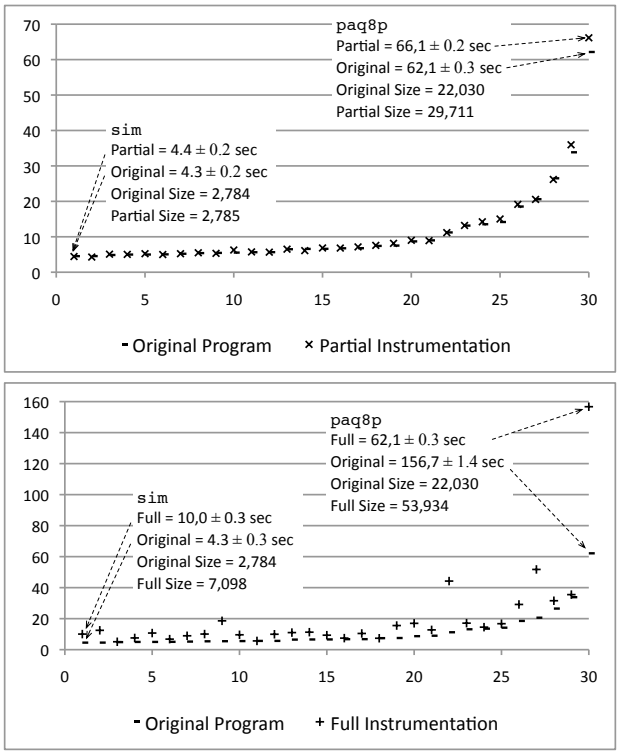


Figure 19. (Top) Runtime of partially instrumented programs (sec). (Bottom) Runtime of fully instrumented programs (sec). Each dot corresponds to a benchmark.

Benchmark	I	P	G	$\gamma \rightarrow$
mcf	4,725	21	1,114	249
libquantum	16,297	25	3,253	770
lbm	13,724	10	3,991	613
astar	19,243	27	5,303	1,327
bzip2	38,831	23	13,150	1,995
milc	44,236	548	17,911	5,605
sjeng	54,051	191	21,282	4,140
hmmer	114,136	50	29,981	6,220
namd	100,76	87	58,300	12,028
soplex	136,364	0	79,424	31,077
h264ref	271,627	86	106,421	14,359
omnetpp	203,201	336	114,329	26,147
gobmk	308,475	163	130,640	27,005
dealII	1,381,408	0	221,305	134,145
xalancbmk	1,314,772	0	774,633	212,377
gcc	1,419,456	41	805,637	119,532

Figure 20. Static Results for the programs in SPEC CPU 2006. P: number of sinks, e.g., `printf`'s in the target program. I: number of x86 instructions produced for the program (about 1.7x larger than the number of bytecodes in the program). G: size of the dependence graph. $\gamma \rightarrow$: number of nodes marked as *maybe* by the forward analysis.

practice. The coefficient of determination between these two quantities, size of the programs and time to run the flow analysis, is 0.92, indicating a strong linear correlation.

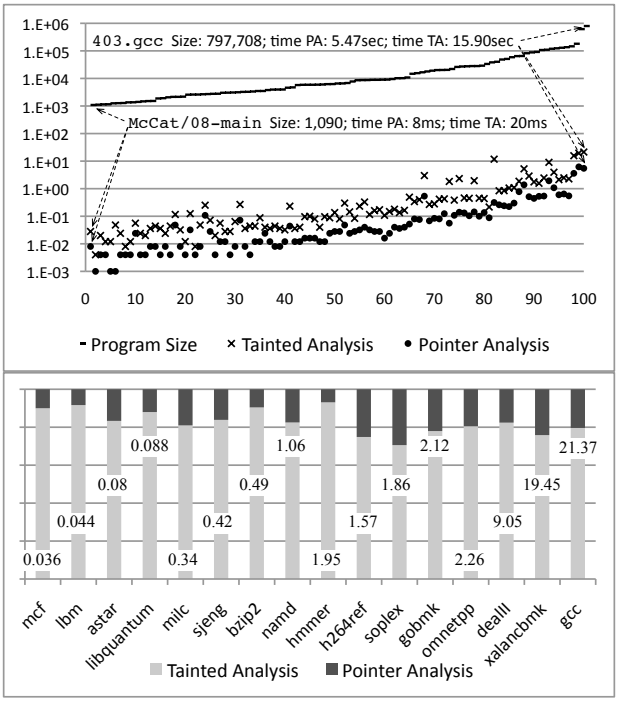


Figure 21. (Top) Time (sec) to run the tainted flow analysis (TA), and the pointer analysis (PA), compared to the size of the programs, given in LLVM bytecode instructions. Each dot in the x-axis represents a benchmark. We considered the 100 largest benchmarks. (Bottom) Time of pointer analysis compared to the time of tainted flow analysis for the programs in SPEC CPU 2006. Numbers give time taken by pointer and flow analysis (secs).

We have also plotted in Figure 21(top) the time to run the points-to analysis, to provide a comparison between our tainted flow analysis and another well-known algorithm, i.e., [15]. On average, the points-to analysis is 3.4 times faster than our tainted flow analysis. Hardekopf's lazy cycle detection is the key feature behind the good performance of our implementation of points-to analysis. Figure 21(bottom) compares the time to run both analysis on the programs in SPEC CPU 2006. In this case, the points-to analysis is 3.73 times faster than our information flow analysis.

The precision of the static analysis. An important question that must be answered is how many warnings produced by the information flow analysis are true-positives. We have obtained only two warnings for the entire SPEC CPU 2006 benchmark suite. A manual inspection of these two warnings reveals that they are true positives. The functions `spec_fread` (at `bzip2/spec.c:187`), and `spec_fwrite` (at `bzip2/spec.c:262`) print the address of buffers that they receive as parameters. Because this code is only executed in debug mode, the dynamic monitor has never flagged leaks in either case. Additionally, we have run our analysis on the complete LLVM test suite, obtaining 19 warnings. We have been able

to recover address information in 11 cases, what gives us a true-positive rate of 11/19, in the universe of 434 programs that we have explored. Our false positives are due to two patterns. The first pattern is subtraction between pointers. As an example, the flow analysis has reported a warning in `Shootout-C++/moments.cc:81` due to the command `printf("n:%d", v.end() - v.begin())`, where `v.end()` and `v.begin()` are pointers. We could not devise a way to recover address information from this code. The second pattern was comparison between pointer and `NULL`. In `2003-05-07-VarArgs.c:59` we got a warning due to the command `printf("LargeS{0x%d}", ls.ptr != 0)`, where `ls.ptr` is a pointer. We have also tried our analysis in a larger application: the implementation of the SpiderMonkey engine used to run JavaScript programs in the Firefox browser version 16.0.1. We found a potential leak at `jscntxtinlines.h:119`. In this case, the command `printf("*** Compartment mismatch %p vs. %p", (void*) c1, (void*) c2)` outputs two addresses.

The paths on dependence graphs between nodes that produce address information and nodes that consume it, e.g., `printf`, tend to be short. The longest path that we have observed has 21 hops, and it represents a false-positive detected in `SIBsim4`. The dependence graph of this benchmark has 1,653 nodes. The shortest path found has only one hop. It is a true positive found in `MallocBench/gs`, whose dependence graph has 6,870 nodes.

V. Related Work

The address disclosure vulnerability is a well-known problem, discussed in blogs and forums. As an example, Dion Blazakis explains how to use pointer inference to compromise an ActionScript JIT compiler [1]. Address leaks are also mentioned by Fermin Serna as a potential vulnerability [2], and the example used in Figure 16 was based on a tutorial available on-line [3]. Nevertheless, the academia has not yet pointed its batteries at this problem, given a lack of publications in the field. However, the techniques that we use in this paper have already been focus of much research. Our work fits in the information flow framework proposed by Denning and Denning [10]. In the words of Hammer and Snelting [14], information flow control deals with two main problems: *integrity* and *confidentiality*. Integrity asks for guarantees that computations cannot be manipulated from outside. Confidentiality asks for guarantees that sensitive data cannot escape the program. In integrity attacks, tainted information flows from an adversary towards sensitive functions; in confidentiality attacks, information flows in the opposite direction. Address disclosures are a confidentiality problem. There exist two main approaches to track the flow of information across a program: dynamic and static analyses. Additionally, engineers can combine these two approaches to obtain faster or more accurate results.

Information flow via dynamic analyses. we recognize four main categories of dynamic analyses: testing, emulation, hardware checks and instrumentation. Testing consists in generating inputs to a program to verify if some events can take place during actual execution. An example of a very influential work in this area is the DART system, which tries, via a combination of testing and symbolic execution, to generate inputs that cover the entire program [13]. Emulation consists in interpreting the program, in order to check if some behavior happens at runtime. A very well-known emulator is the Valgrind tool [20], which supports different kinds of checks that can be performed on the running program. We also classify Perl's *taint mode*, used to prevent tainted flow attacks, as an emulation-based dynamic analysis [1]. The third approach, hardware checks, consists in tracking the propagation of information at the architectural level [31].

Finally, instrumentation consists in adding to the program extra code that captures events of interest. In other words, this approach requires the compiler to modify the program. Such program will run directly in native mode, instead of being interpreted. The instrumentation framework that we have designed and implemented fits in this last category. Instrumentation is generally used in two different ways: at the machine independent level, or at the binary level. This paper uses a machine independent instrumentation library. Other machine independent approaches include Xu's taint checker [35], and Dietz *et al.*'s integer overflow checker [11]. The literature contains a plethora of works at the binary, i.e., machine dependent level. For a general overview of these approaches we recommend the Related Work sections in Clause's [8] and Newsome's [21] papers. The main advantage of binary instrumentation is the ability to deal precisely with code from external libraries. The disadvantages include a higher runtime overhead, and the nontrivial integration with static analyses. This latter shortcoming, difficult integration with static analysis, was the key factor that motivate us to stay at the compiler's intermediate representation level.

Information flow via static analyses. Most of the literature related to information flow relies on static analysis as the tool of choice to track information [17], [23], [25], [33], [34]. The vast majority of these works are concerned with integrity, i.e., "can malicious data flow into some sensitive program operation?" Yet, there are also works that focus on confidentiality [14], [26]. There are many different approaches to describe static analyses of information flow: type systems [14], [16], data-flow [17], abstract interpretation [4] and program slicing [25]. Nevertheless, all these papers - ours included - seem to rely on similar algorithms that seek dangerous paths on the program's dependence graph [12].

We chose to formalize our static analysis as a type system, because in our opinion this approach makes it easier to

[1] `http://www.semantiscope.com/research/BHDC2010`
[2] *The info leak era on software exploitation*, slides available on-line.
[3] `Pwn2Own-2010-Windows7-InternetExplorer8.pdf`

provide correctness proofs. Our static analysis combines a forward and a backward component to obtain more precise results. To the best of our knowledge, the only work that also combines a backward and a forward analyses is Rimsa *et al.*'s [25]. However, they rely on a graph reacheability formulation that does not make this approach explicit. Moreover, whereas Rimsa *et al.*'s work is concerned about integrity - they try to secure programs against XSS attacks - we focus on confidentiality.

The combination of static and dynamic analyses. Many different researchers have combined dynamic and static analyses to track information along the program pathways. This combination is motivated by either speed or precision. In the latter case, the static analysis complements the information that is acquired by its dynamic counterpart, or the dynamic analysis eliminates false-positives produced by the static algorithm. For instance, Zhang *et al.* [37] have proposed a tool that first instruments the binary program and runs it, searching for tainted flow vulnerabilities. During this guarded execution, the tool also builds the control flow graph of the program. This structure is later given to a static analyzer, that performs a second round of program verifications, conservatively checking the program parts that have not been reached by the execution flow. Vogt *et al.* [32] also use static analysis to extend the results of the dynamic analysis; however, they do it *on the fly*. That is, once the dynamic analysis identifies program regions that can be controlled by tainted data, Vogt *et al.*'s tool performs a linear pass on that region, tainting all the variables defined inside it. In a different direction, Balzarotti *et al.* [2] have designed Saner, a tool that uses test-based dynamic analysis to check the possible vulnerabilities reported by the static analysis.

Static analysis can also be used to improve the runtime of the dynamic technique. Possibly the most influential work in this area is Huang *et al.*'s WebSSARI [16]. This tool uses a type system to identify tainted program variables, and inserts sanitizers at the sensitive functions that read these variables. Our work is different from Huang's in several dimensions. First, we deal with confidentiality, they deal with integrity. Second, our type system is bidirectional, theirs is unidirectional (forward). Third, they instrument only sensitive functions, whereas we instrument the entire tainted path, because in our case the notion of sanitizer is not well-defined. In a totally opposite direction to Huang's work, and ours, but still motivated by efficiency, Chebaro *et al.* [6] have used a static analysis to slice secure parts of a program away. Then they use test-based dynamic analysis on the reduced program to search for vulnerabilities in its unsafe parts.

VI. Conclusion

This paper has discussed the address disclosure problem. This vulnerability allows adversaries to circumvent Address Space Layout Randomization, a protection mechanism used in popular operating systems. We showed via an example how address leaks can be explored in practice, and formalized this vulnerability via a simple, yet Turing complete, language. In addition of introducing the problem, we showed how such vulnerabilities can be statically uncovered, and dynamically prevented. The combination of the static and the dynamic analyses let us produce safe code that suffers almost no runtime overhead. Additionally, we have used our type system to detect actual address leaks in very large programs.

Future works: we plan to extend this work in two different directions. Firstly, the main purpose behind our instrumentation framework is to support the detection of address leaks via automatic test generation. Thus, we are currently working to integrate our instrumentation library with a publicly available test generator. Secondly, although we are able to correctly instrument large programs, our framework is a research artifact. Consequently, some partially instrumented codes are still much slower than the original programs. To decrease this overhead, we want to use optimizations such as Qin's style code coalescing [24], a technique that has been developed to improve the runtime of binary instrumentation frameworks.

Reproducibility: All the benchmarks used in Section IV, other than SPEC CPU 2006, are publicly available in the LLVM website. The LLVM compiler is open source. Our code and further material about this project, including a Prolog interpreter of Angels, is available at the anonymous website `http://code.google.com/p/addr-leaks/`.

References

[1] J. Allen. *Perl*. Perlsec, 5.8.8 edition, 2006.

[2] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *SP*, pages 387–401. IEEE Computer Society, 2008.

[3] Eep Bhatkar, Daniel C. Duvarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *USENIX Security*, pages 105–120, 2003.

[4] B. Blanchet, P Cousot, R. Cousot, J. Feret, L. Mauborgne, A Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207. ACM, 2003.

[5] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In *CCS*, pages 27–38. ACM, 2008.

[6] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. Program slicing enhances a verification technique combining static and dynamic analysis. In *SAC*, pages 1284–1291. ACM, 2012.

[7] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *POPL*, pages 55–66. ACM, 1991.

[8] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA*, pages 196–206. ACM, 2007.

[9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.

[10] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20:504–513, 1977.

[11] Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding integer overflow in c/c++. In *ICSE*, pages 760–770. IEEE, 2012.

[12] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319–349, 1987.

[13] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223. ACM, 2005.

[14] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, 2009.

[15] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI*, pages 290–299. ACM, 2007.

[16] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *WWW*, pages 40–52. ACM, 2004.

[17] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Security & Privacy*, pages 258–263. IEEE, 2006.

[18] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE, 2004.

[19] Elias Levy. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.

[20] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100. ACM, 2007.

[21] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*. USENIX, 2005.

[22] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *PLDI*, pages 257–271. ACM, 1990.

[23] Marco Pistoia, Robert J. Flynn, Larry Koved, and Vugranam C. Sreedhar. Interprocedural analysis for privileged code placement and tainted variable detection. In *ECOOP*, pages 362–386, 2005.

[24] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO*, pages 135–148. IEEE, 2006.

[25] Andrei Alves Rimsa, Marcelo D'Amorim, and Fernando M. Q. Pereira. Tainted flow analysis on e-SSA-form programs. In *CC*, pages 124–143. Springer, 2011.

[26] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *CSF*, pages 186–199. ACM, 2010.

[27] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and Privacy*, pages 317–331. IEEE, 2010.

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

[29] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *CSS*, pages 298–307. ACM, 2004.

[30] Bjarne Stroustrup. Evolving a language in and for the real world: C++ 1991-2006. In *HOPL*, pages 1–59. ACM, 2007.

[31] G. Suh, J. Lee, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS*, pages 85–96. ACM, 2004.

[32] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, 2007.

[33] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI*, pages 32–41. ACM, 2007.

[34] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS*. USENIX Association, 2006.

[35] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security*. USENIX, 2006.

[36] Zhichen Xu, Barton P. Miller, and Oscar Naim. Dynamic instrumentation of threaded applications. In *PPoPP*, pages 49–59. ACM, 1999.

[37] Ruoyu Zhang, Shiqiu Huang, Zhengwei Qi, and Haibin Guan. Combining static and dynamic analysis to discover software vulnerabilities. In *IMIS*, pages 175–181. IEEE Computer Society, 2011.