Static Detection of Address Leaks

Gabriel Quadros Silva and Fernando Magno Quintão Pereira

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

{gabrielquadros,fpereira}@dcc.ufmg.br

Abstract. *Taint analysis is a form of program analysis that determines if values produced by unsafe sources might flow into sensitive functions. In this paper we use taint analysis to establish if an adversary might discover the address of any program variable at runtime. The knowledge of an internal program address seems, in principle, a harmless information; however, it gives a malicious user the means to circumvent a protection mechanism known as address space layout randomization, typically used in modern operating systems to hinder buffer overflow attacks, for instance. We depart from previous taint analyses because we also track indirect information leaks, in which confidential data is first stored in memory, from where it flows into some sensitive operation. We have implemented our analysis into the LLVM compiler and have used it to report 204 warnings in a test suite that contains over 1.3 million lines of C code, and includes traditional benchmarks such as SPEC CPU 2006. Our current implementation reduces by more than 14 times the number of sensitive operations that a developer would have to inspect in order to find address leaks manually. Furthermore, our analysis is remarkably efficient: it has been able to process more than 8.2 million assembly instructions in 19.7 seconds!*

1. Introduction

There seems to exist an "arms race" between program developers and malicious users, or *crackers*, as they are popularly called. Every day we hear about new strategies that are invented to attack sensitive software, and every day we hear about new security mechanisms that are engineered to protect these systems. Buffer overflows are a very well known technique that untrusted code uses to compromise other programs. Its basic principle consists in writing on an array a quantity of data large enough to go past the array's upper bound; hence, overwriting other program data. The Internet Worm of 1988, probably the most famous case of viral spreading of malicious software in the Internet, exploited a buffer overflow in the `fingerd` daemon [Rochlis and Eichin 1989]. To prevent buffer overflows exploits, operating system engineers have invented a technique called *address space layout randomization* [Bhatkar et al. 2003, Shacham et al. 2004], that consists in loading some key areas of a process at random locations in its address space. In this way, the attacker cannot calculate precisely the target addresses that must be used to take control of the vulnerable program.

However, crackers are able to circumvent the address randomization mechanism, as long as they can have access to an internal program address. Armed with this knowledge, malicious users can estimate the exact base address of the functions available to the executing program, an information that gives them a vast suite of possibilities to compromise this program [Levy 1996]. A cracker can discover an internal program address in

many different ways. For instance, many applications contain debugging code that dumps runtime information, including variable addresses. By using the correct flags, the attacker may easily activate this dumping. Fancier strategies, of course, are possible. In some object oriented systems the hash code of an object is a function of the object's address. If the hash-function admits an inverse function, and this inverse is known, then the attacker may obtain this address by simply printing the object's hash code.

The objective of this paper is to describe a static code analysis that detects the possibility of an address information leaking from a program. Our technique is a type of *taint analysis* [Rimsa et al. 2011], that is, given a set of source operations, and a set of sink operations, it finds a flow of information from any source to any sink. We differ from previous works in two ways: first, we are proposing a novel use of taint analysis; second, we deal with *indirect leaks*. Concerning the first difference, the leaking of address information is a problem well known among software engineers, as a quick glance at blogs related to computer security would reveal [1]. Nevertheless, in spite of the importance of this problem, the research community has not yet pointed its batteries at it, as we can infer from a lack of publications in the field. In addition to exploring a new use of taint analysis, we extend the information flow technology with a method to track indirect leaks. An indirect leak consists in storing sensitive information in memory, and then reading this information back into local program variables whose contents eventually reach a sink operation. As recently discussed in the USENET [2], developers and theoreticians alike avoid having to track information through the memory heap because it tends to be very costly in terms of processing time. However, by relying on a context insensitive interprocedural analysis we claim to provide an acceptable tradeoff between efficiency and precision.

We have implemented our analysis on top of the LLVM compiler [Lattner and Adve 2004], and have used it on a collection of C programs comprising over 1.3 million lines of code. This test suite includes well-known benchmarks such as SPEC CPU 2006, Shootout and MediaBench. Our implementation has reported 204 potential address leaks. We have manually inspected 16 reports taken from the 16 largest programs in our benchmark suite, and have been able to recover 2 actual program addresses. Although this number seems low, we remark that our analysis reduced by 14 times the number of sensitive statements that a developer would have to verify in order to find address leaks. Our implementation is very efficient: it takes about 19.7 seconds to process our entire test suite – a collection of programs having over 8.26 million assembly instructions. As an example, in order to analyze `gcc`, a well known member of SPEC CPU 2006, with 1,155,083 assembly instructions, our implementation takes 2.62 seconds.

The rest of this paper is organized in five other sections. In Section 2 we explain in more details why address leaks enable malicious users to successfully attack programs. In Section 3 we introduce our solution and discuss its limitations. We show experimental results in Section 4. Section 5 discusses several works that are related to ours. Finally, Section 6 concludes the paper.

[1] `http://mariano-graziano.llab.it/docs/stsi2010.pdf`
`http://www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf`
`http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf`

[2] `http://groups.google.com/group/comp.compilers/browse_thread/thread/1eb71c1177e2c741`

2. 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 [Stroustrup 2007], is the reason behind an uncountable number of worms and viruses that spread on the Internet [Bhatkar et al. 2003].

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.

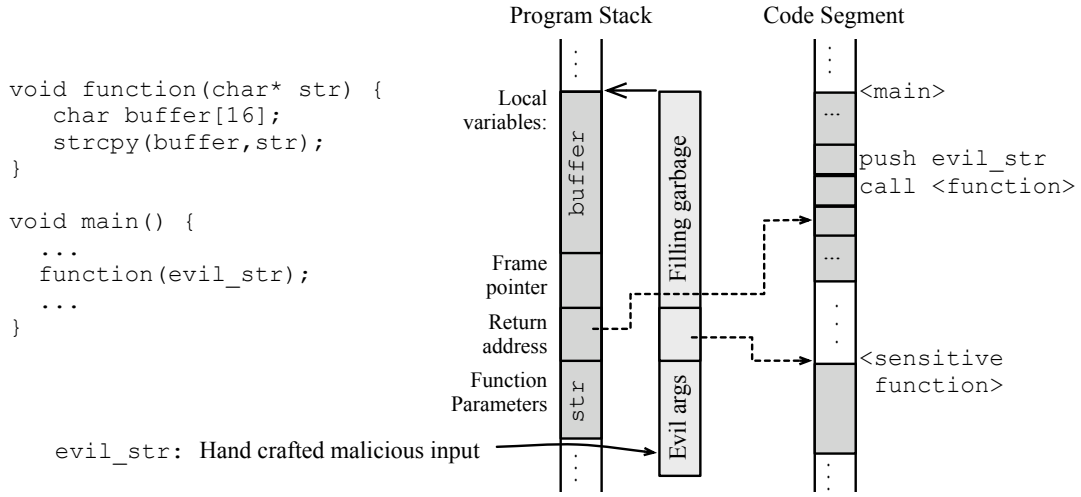

Figure 1. An schematic example of a stack overflow. The return address of `function` is diverted by a maliciously crafted input to another procedure.

A *buffer overflow* consists in writing in a buffer a quantity of data large enough to go past the buffer's 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. The first buffer overflow attacks included the code that should be executed in the input array [Levy 1996]. However, modern operating systems mark writable memory addresses as non-executable – a protection mechanism

known as *Read⊕Write* [Shacham et al. 2004, p.299]. Therefore, attackers tend to divert execution to operating 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. This type of attack is called *return to* `libc` [Shacham et al. 2004]. Return to `libc` attacks have been further generalized to a type of attack called *return-oriented-programming* (ROP) [Shacham 2007]. If a binary program is large enough, then it is likely to contain many bit sequences that encode valid instructions. Hovav Shacham [Shacham 2007] has shown how to derive a Turing complete language from these sequences in a CISC machine, and Buchanan *et al.* [Buchanan et al. 2008] 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* [Bhatkar et al. 2003]. 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.* [Shacham et al. 2004] have shown that these methods are susceptible to brute force attacks; nevertheless, address obfuscation slows down the propagation rate of worms that rely on buffer overflow vulnerabilities substantially. Address obfuscation is not, however, the ultimate defense mechanism. 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 detect in this paper.

3. The Proposed Solution

We detect address leaks via a three steps process. Firstly, we convert the program to a suitable *normal form*, in which every language construct that is interesting to us is converted to a few *constraints*. Secondly, we build a *dependence graph* out of the constraints previously defined. Finally, we perform a depth-first search on this dependence graph to report leaks. We explain in more details each of these steps in this section.

3.1. Converting the Source Program to a Normal Form

We use a constraint system to detect address leaks. In order to represent the five different types of constraints that we take into consideration, we define a simple constraint language, whose syntax is given in Figure 2. We produce these constraints out of actual C or C++ programs, as the table in Figure 3 illustrates. We use `getad` 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`. Program expressions that do not include any memory address are modeled via the constraint `simop`, a short name for *simple operation*. Loads to and stores from memory are modeled by `ldmem` and `stmem`. Finally, we use `print` to denote any instruction that gives information away to an external user. This constraint models not only ordinary printing operations, but

(Variables)	::=	$\{v_1, v_2, \ldots\}$
(Constraints)	::=	
– (Assign variable address)	|	$\texttt{getad}(v_1, v_2)$
– (Simple variable assignment)	|	$\texttt{simop}(v, \{v_1, \ldots, v_n\})$
– (Store into memory)	|	$\texttt{stmem}(v_0, v_1)$
– (Load from memory)	|	$\texttt{ldmem}(v_1, v_0)$
– (Print the variable's value)	|	$\texttt{print}(v)$

Figure 2. The syntax of our constraint system.

`v1 = &v2`	$\texttt{getad}(v_1, v_2)$
`v1 = (int*)malloc(sizeof(int))`	$\texttt{getad}(v_1, v_2)$ where v_2 is a fresh memory location
`v1 = *v0`	$\texttt{ldmem}(v_1, v_0)$
`*v0 = v1`	$\texttt{stmem}(v_0, v_1)$
`*v1 = *v0`	$\texttt{ldmem}(v_2, v_0)$, where v_2 is fresh $\texttt{stmem}(v_1, v_2)$
`v = v1 + v2 + v3`	$\texttt{simop}(v, \{v_1, v_2, v_3\})$
`*v = v1 + &v2`	$\texttt{getad}(v_3, v_2)$, where v_3 is fresh $\texttt{simop}(v_4, \{v_1, v_3\})$, where v_4 is fresh $\texttt{stmem}(v, v_4)$
`f(v1, &v3)`, where `f` is declared as `f(int v2, int* v4);`	$\texttt{simop}(v_2, \{v_1\})$ $\texttt{getad}(v_4, v_3)$

Figure 3. Examples of mappings between actual program syntax and constraints.

also native function interfaces, which would allow a malicious JavaScript file to obtain an internal address from the interpreter, for instance.

We have designed our analysis to work on programs in *Static Single Assignment* form. This is a classic compiler intermediate representation [Cytron et al. 1991] in which each variable name is defined only once. Virtually every modern compiler today uses the SSA form to represent programs internally, including Java HotSpot [Team 2006], gcc [Gough 2005] and LLVM [Lattner and Adve 2004], the compiler on top of which we have implemented our algorithms. The single static assignment property, i.e., the fact that each variable name is unique across the entire program, is essential to allow us to bind to each variable the state of being *trusted* or *untrusted*. Because we provide an interprocedural analysis, i.e., we analyze whole programs, we assume *global SSA form*. In other words, each variable name is unique in the entire program, not only inside the scope where that variable exists. In practice we obtain global SSA form by prefixing each variable name with the name of the function where that variable is defined.

[EMPTY] $\texttt{build\_edges}(\varnothing, P_t) = \varnothing$

[EDGES] $$\frac{\texttt{build\_edges}(C, P_t) = E \qquad \texttt{proc\_con}(c, P_t) = E'}{\texttt{build\_edges}(C \cup \{c\}, P_t) = E \cup E_\iota}$$

[GETAD] $\texttt{proc\_con}(\texttt{getad}(v_1, v_2), P_t) = \{\texttt{val}(v_1) \rightarrow \texttt{addr}(v_2))\}$

[PRINT] $\texttt{proc\_con}(\texttt{print}(v), P_t) = \{\texttt{sink} \rightarrow \texttt{val}(v)\}$

[SIMOP] $\texttt{proc\_con}(\texttt{simop}(v, \{v_1, \ldots, v_n\}), P_t) = \{\texttt{val}(v) \rightarrow \texttt{val}(v_i) \mid 1 \leq i \leq n\}$

[STMEM] $\texttt{proc\_con}(\texttt{stmem}(v_0, v_1), P_t) = \{\texttt{val}(v) \rightarrow \texttt{val}(v_1) \mid v_0 \mapsto v \in P_t\}$

[LDMEM] $\texttt{proc\_con}(\texttt{ldmem}(v_1, v_0), P_t) = \{\texttt{val}(v_1) \rightarrow \texttt{val}(v) \mid v_0 \mapsto v \in P_t\}$

Figure 4. Recursive definition of the edges in the memory dependence graph.

3.2. Building the Memory Dependence Graph

Once we extract constraints from the target C program, we proceed to build a *memory dependence graph*. This – directed – graph is a data structure that represents the patterns of dependences between variables. If P is a target program, and $G = (V, E)$ is P's dependence graph, then for each variable $v \in P$ we define two vertices: a *value* vertex, which we denote by $\texttt{val}(v) \in V$ and an *address* vertex, which we represent by $\texttt{addr}(v) \in V$. We say that location v_1 depends on location v_0 if v_0 is necessary to build the value of v_1. In actual programs such dependences happen any time v_0 denotes a value used in an instruction that defines v_1, or, recursively, v_0 denotes a value used in an instruction that defines a variable v_2 such that v_1 depends upon v_2.

More formally, given a set C of constraints that follow the syntax in Figure 2, we define the memory dependence graph via the function `build_edges`, shown in Figure 4. The only constraint that produces edges pointing to address nodes is `getad`, as we show in Rule GETAD in Figure 4. If v_1 is defined by an instruction that reads the address of variable v_2, then we insert an edge $\texttt{val}(v_1) \rightarrow \texttt{addr}(v_2)$ into E. The memory dependence graph has a special node, which we call `sink`. Edges leaving `sink` towards value nodes are created by Rule PRINT. From a quick glance at Figure 4 it is easy to see that `sink` will have in-degree zero. Rule SIMOP determines that we generate an edge from the value node that represents the variable defined by a simple operation towards the value node representing every variable that is used in this operation. In other words, if v_1 is defined by an instruction that reads the value of v_2, then we insert an edge $\texttt{val}(v_1) \rightarrow \texttt{val}(v_2)$ into our memory dependence graph.

$$[\textsc{Leak}] \quad \frac{\texttt{build\_edges}(C, P_t) = E \qquad \texttt{dfs}(\texttt{sink}, E) = B}{\texttt{find\_leak}(C, P_t) = B}$$

$$[\textsc{Sink}] \quad \frac{\texttt{sink} \rightarrow \texttt{val}(v) \in E \qquad \texttt{dfs}(v, E) = B}{\texttt{dfs}(\texttt{sink}, E) = B}$$

$$[\textsc{Val}] \quad \frac{\texttt{val}(v) \rightarrow \texttt{val}(v') \in E \qquad \texttt{dfs}(v', E) = B}{\texttt{dfs}(v, E) = B \cup \{\texttt{val}(v_1) \rightarrow \texttt{addr}(v_2)\}}$$

$$[\textsc{Addr}] \quad \frac{\texttt{val}(v_1) \rightarrow \texttt{addr}(v_2) \in E}{\texttt{dfs}(v, E) = \{\texttt{val}(v_1) \rightarrow \texttt{addr}(v_2)\}}$$

Figure 5. Recursive definition of an address leak.

The processing of load and store constraints is more complicated, because it demands *points-to* information. We say that a variable v_1 points to a variable v_2 if the value of v_1 holds the address of v_2. The problem of conservatively estimating the points-to relations in a C-like program has been exhaustively studied in the compiler literature [Andersen 1994, Hardekopf and Lin 2007, Pereira and Berlin 2009, Steensgaard 1996]. Therefore, we assume that we start the process of building the memory dependence graph with a map $P_t : V \mapsto PowerSet(V)$ that tells, for each variable $v \in V$, what is the subset of variables $V' \subseteq V$ such that v points to every element $v' \in V'$. According to Rule STMEM, whenever we store a variable v_1 into the address pointed by variable v_0, i.e., in the C jargon: `*v0 = v1`, then, for each variable v pointed by v_0 we create an edge from the value node of v towards the value node of v_0. The `ldmem` constraint works in the opposite direction. Whenever we load the value stored in the address pointed by v_0 into a variable v_1, i.e., `v1 = *v0`, then, for each variable v that might be pointed-to by v_0 we add an edge from the value node of v_1 to the value node of v.

3.3. Traversing the Memory Dependence Graph to Find Address Leaks

Figure 5 defines a system of inference rules to characterize programs with address leaks. This definition also gives a declarative algorithm to find a path B in the memory dependence graph describing the address leak. Rule LEAK tells us that a constraint system C, plus a set of points-to facts P_t describes at least one address leak if the memory dependence graph built from C and P_t has a set of edges E, and E contains a path B, from `sink` to an address node. To denote this last statement, we use the `dfs` predicate, which describes a depth-first search along E, as one can readily infer from the Rules SINK, VAL and ADDR. These rules are self explanatory, and we will not describe them further.

3.4. An Example of our Analysis in Action

We illustrate the concepts introduced in this section via the C program shown in Figure 6. This program, although very artificial, contains the main elements that will allows us to

```
1  int g(int p) {
2    int** v0 = (int**)malloc(8);       // getad(v0, m1)
3    int* t0 = (int*)malloc(4);         // getad(t0, m2)
4    *t0 = 1;
5    *v0 = t0;                          // stmem(v0, t0)
6    while (p > 1) {
7      int* v1 = *v0;                   // ldmem(v1, v0)
8      int t1 = *v1;                    // ldmem(t1, v1)
9      printf("%d\n", t1);              // print(t1)
10     int* v2 = (int*)malloc(4);       // getad(v2, m3)
11     int* t2 = *v0;                   // ldmem(t2, v0)
12     *t2 = (int)v2;                   // stmem(t2, v2)
13     p--;
14   }
15 }
```

Figure 6. A C program that contains an address leak: variable `t1` might contain the address of the memory region allocated at line 10.

illustrate our analysis. The constraints that we derive from the program, as explained in Section 3.1, are given as comments on the right side of Figure 6. Let's assume, for the sake of this example, that variable `v0` points to a memory region `m1`, created in line 2 of Figure 6. We also assume that variables `v1` and `t2` point to a memory region `m2`, created in line 10 of our example. These points-to facts are computed beforehand, by any standard alias analysis implementation, as we have explained in Section 3.2. Figure 7(a) shows, again, the constraint set C that we must process for the example in Figure 6, and Figure 7(b) re-states the points-to facts that are known before we start our analysis.

Once we have converted the target program to a normal form, we must build its memory dependence graph, according to the rules in Figure 4 from Section 3.2. Figure 7(c) shows the graph that we build for this example. We chose to use a particular notation to represent the nodes. Each variable v gives origin to two vertices, e.g. $\texttt{val}(v)$ and $\texttt{addr}(v)$; hence, each vertex in our graph is represented as the juxtaposition of two boxes. The first, denoting the value node, contains the name of the variable it represents, whereas the second box – containing an @ – represents this variable's address. Our example graph contains nine such nodes, one for each variable defined in the target program, plus a special node, depicted as a black diamond (◆), which represents the `sink`.

Once we have built the memory dependence graph, the next step is to traverse it, reporting unsafe paths. We perform this last step using the rules in Figure 5, as explained in Section 3.3. The program in Figure 6 contains an address leak, which is easy to find in the graph from Figure 7. The problematic path is $\texttt{sink} \rightarrow \texttt{val}(t_1) \rightarrow \texttt{val}(m_2) \rightarrow \texttt{val}(v_2) \rightarrow \texttt{addr}(m3)$. Going back to Figure 6, this path corresponds to printing the value of `t1`. In order to see why this output is an address leak, notice that `t1`, `*v1`, `**v0` and `*t2` might represent the same value, which, as we see in line 12 of our example, is the address of the memory location pointed by `v2`.

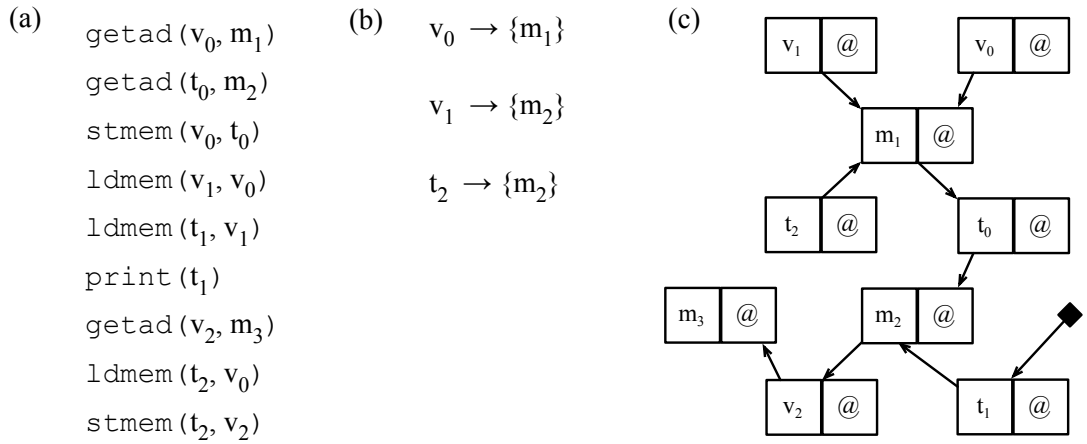

Figure 7. (a) The constraint system derived from the Program in Figure 6. (b) Points-to facts, computed beforehand via Andersen's analysis [Andersen 1994]. (c) The memory dependence graph built from the constraints and points-to facts.

3.5. Limitations

The current implementation of our analysis has two main limitations. First it is context insensitive, which means that we cannot distinguish two different calls from the same function. Second, it is field and array insensitive; hence, objects, records and arrays are treated as a single memory unit. These limitations lead us to report warnings that are *false positives*, or that, in other words, represent innocuous program patterns.

Our analysis is *interprocedural*, which means that we can track the flow of information across function calls. However, our analysis is *context insensitive*, that is, we cannot distinguish different invocations of the same procedure. As an example, the program in Figure 8 does not contain an address leak. Nevertheless, the function calls at lines 9 and 13 leads us to link the contents of `v0` to `v3`, even though these variables are never related in the actual program semantics. Because `v0` contains a program address, and `v3` is printed, we issue a warning. As a future work, we plan to improve our framework with light-weighted context sensitive methods, such as those based on probabilistic calling contexts [Bond and McKinley 2007] or shallow heap cloning [Lattner et al. 2007].

Our second limitation is a lack of field sensitiveness. We treat programming language constructs, such as objects, records and arrays as single locations. Figure 9 contains an example of a bug free program that causes us to issue a warning. The assignment in line 7 marks the whole variable `s1` as tainted. Therefore, even the innocuous printing statement at line 9 is flagged as a possible leak. As a future work, we intend to extend our analysis with Pearce *et al.*'s [Pearce et al. 2004] field sensitive constraint system.

4. Experimental Results

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. We have tested our algorithm on a collection of 426 programs written in C that we got from the LLVM test suite. In total, we have analyzed 8,427,034

```
1  int addSizeInt(int n0) {
2    int n1 = n0 + sizeof(int);  // simop(n1, n0)
3    return n1;
4  }
5  int main() {
6    int* v0 = (int*)
       malloc(2 * sizeof(int));  // getad(v0, m1)
7    int* v1;
8    int v2 = 0, v3 = 1, v4 = 1;
9    v1 = addSizeInt(v0);        // simop(n0, v0), simop(v1, n1)
10   *v1 = v4;                   // stmem(v1, v4)
11   int v5 = *v1;               // ldmem(v5, v1)
12   printf("%d\n", v5);         // print(v5)
13   v3 = addSizeInt(v2);        // simop(n0, v2), simop(v3, n1)
14   printf("%d\n", v3);         // print(v3)
15 }
```

Figure 8. The lack of context sensitiveness in our analysis will cause us to report a false positive for this program.

```
1  struct S {
2    int harmless;
3    int dang;
4  };
5  int main() {
6    struct S s1;
7    s1.harmless = (int)&s1;     // getad(s1, s1)
8    s1.dangerous = 0;
9    printf("%d\n", s1.dang);     // print(s1)
10 }
```

Figure 9. The lack of field sensitiveness in our analysis cause us to report a false positive for this program.

assembly instructions. We will present results for SPEC CPU 2006 only, which is our largest benchmark suite. Table 1 gives details about each of the 17 programs in the SPEC collection. Without loss of generality, for these experiments we qualify as sensitive sinks the standard `printf` operation from the `stdio.h` header. In other words, we are seeking for dependence chains that cause an internal program address to be printed by a `printf` function. There exist other functions that may lead to address leaking. Our tool can be configured to check these functions by marking (i) return statements and (ii) assignments to parameters passed as references, as sink operations.

We will compare three different configurations of our address leak detector, which we call *Direct*, *MDG* and *Blob*. The first approach does not track information through memory. That is, it only reports the propagation of information through local program variables. The second approach – MDG – uses the Memory Dependence Graph that we have described in Section 3.2 to track the flow of information through memory. Finally, the third method – Blob – assumes that the whole program memory is a single, indivisible unit. In this case, any operation that stores the value of an address into memory

will contaminate the whole heap and stack. If any information from the tainted memory posteriorly flows into a sink, we will issue a warning.

Precision: Table 1 shows the number of warnings that our tool has reported per program in SPEC CPU 2006. The table reveals a wide contrast between the Direct and Blob approaches. In the former case, every warning turns out to be a true positive that allows us to recover an internal program address. However, the direct method misses many leaks that the other two approaches are able to point out. The blob technique, on the other hand, contains too many *false positives*, that is, a substantial number of warnings that it reports are actually innocuous. MDG is a compromise: it finds every true positive pointed by blob, and omits many false positives. A manual inspection of the first warning reported by MDG for each benchmark gave us a 1/8 false positive rate. The false positives are caused by the limitations described in Section 3.5, which we are working to overcome. Nevertheless, MDG reduces by 14x, on average, the number of `printf` statements that a developer would have to verify in order to find potential address leaks. The chart in Figure 10 puts this number in perspective, showing, for each benchmark and tracking method, the percentage of printing statements that are flagged as potential address leaks.

Benchmark Program	**Number of Instructions**	**Number of** `printf`'s	Warnings			Time (msec)		
			Blob	MDG	Direct	Blob	MDG	Direct
`mcf`	4005	12	8	0	0	36	12	8
`lbm`	5522	8	2	0	0	16	12	1
`libquantum`	11422	30	28	5	0	168	56	16
`astar`	14228	14	8	8	0	64	64	20
`bzip2`	24881	21	21	5	0	220	88	28
`sjeng`	34474	88	40	0	0	704	52	44
`milc`	35357	191	86	16	0	1200	252	48
`hmmer`	98150	52	17	0	0	508	184	120
`soplex`	119616	0	0	0	0	172	260	176
`namd`	121065	18	10	0	0	344	196	128
`h264ref`	176652	53	19	1	0	932	320	220
`omnetpp`	199934	20	7	5	1	624	604	308
`gobmk`	222071	64	19	2	0	2792	696	316
`perlbench`	388436	0	0	0	0	576	760	500
`dealII`	934844	3	0	0	0	1680	2128	1476
`gcc`	1155083	16	3	0	0	2628	2252	1632
`xalancbm`	1428459	8	7	1	1	5516	3568	106

Table 1. Summary of main experimental results for SPEC CPU 2006.

Running time: The three versions of the address leak analysis are very fast. The direct approach took 5,147 msecs to process SPEC CPU 2006. Blog took 18,180 msecs, and MDG 11,504 msecs. Furthermore, MDG took 19.7 seconds to analyze the entire LLVM test suite plus SPEC CPU 2006, a benchmark collection that gives us over 8.26 million assembly instructions! The three analyses show a linear complexity behavior in practice. The charts in Figure 11 shows MDG's processing time for programs in our benchmark collection having more than 20,000 assembly instructions. These 38 programs, from the LLVM test suite plus SPEC CPU 2006, contain over 7.64 million assembly instructions.

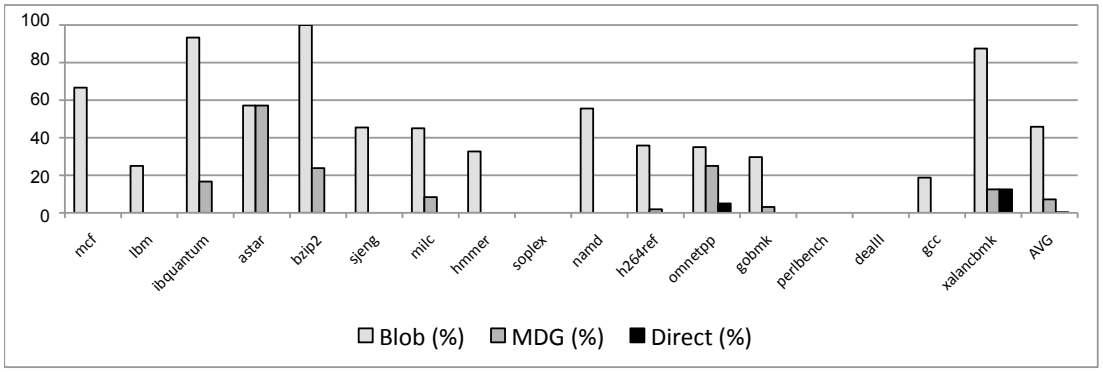

Figure 10. Percentage of `printf` statements flagged as potential address leaks.

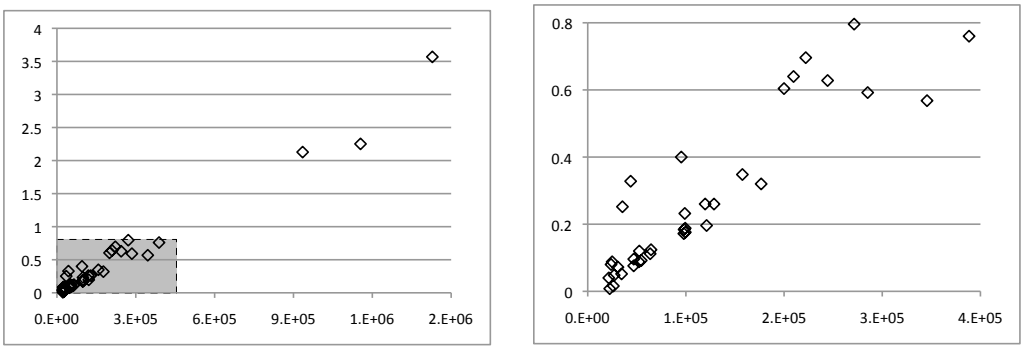

Figure 11. Size Vs Time: each dot represents a benchmark program. X axis: number of instructions. Y axis: time to analyze using MDG (msec). The chart on the right is a close up of the gray area on the chart in the left.

A visual inspection of the chart indicates that the processing time grows linearly with the program size. We have also observed this tendency in the smaller programs.

5. Related Works

The problem that we deal with in this paper – the leaking of an internal program address – fits in the *information flow framework* proposed by Denning and Denning [Denning and Denning 1977]. A program address is the information that we want to track, and the program is deemed safe if this information cannot flow into an output operation. The algorithm that we propose to detect address leaks is a type of tainted flow analysis. Similar analysis have been proposed in the literature before, to detect, for instance, if malicious data that a user feeds to some input function can flow into some sensitive program operation [Jovanovic et al. 2006, Pistoia et al. 2005, Rimsa et al. 2011, Wassermann and Su 2007, Xie and Aiken 2006]. None of these previous works handle indirect information flows through memory, like we do. Furthermore, none of them track address leaking. Instead, these analyses uncover vulnerabilities to exploits such as SQL injection or cross site scripting attacks.

Our memory dependence graph is similar to the shape graphs used in *shape analysis* [Sagiv et al. 1998, Sagiv et al. 2002]. However, whereas in shape analysis one such

graph is built for each program point, i.e, the region between two consecutive assembly instructions, we use only one graph for the whole program. Therefore, shape analysis gives the compiler much more precise knowledge about the memory layout of the program; however, its high cost, both in time and space, causes it to be prohibitively expensive to be used in practice. Still concerning the representation of memory locations, Ghiya and Hendren [Ghiya and Hendren 1998] have proposed an algorithm that relies on points-to information to infer disjoint data-structures. We could, in principle, use their technique to track information leaks through memory location, but it would be more conservative than our current approach, for we can track different memory cells used inside the same data-structure. Our problem is also related to *escape analysis* [Blanchet 1998], which conservatively estimates the set of memory locations that outlive the function in which these locations have been created. The address leaking problem is more general, because we track the flow of addresses inside or across functions.

6. Conclusion

This paper has presented a static analysis tool that checks if an adversary can obtain the knowledge of an internal program address. This is a necessary step in order to circumvent a program protection mechanism known as address space layout randomization. We have implemented our algorithms on top of LLVM, an industrial strength compiler, and have used it to process a collection of programs with more than 1.3 million lines of C code. We have been able to discover actual address leaks in some of these programs. Currently we are working to reduce the number of false positives reported by our implementation. We plan to do it by adding context and field sensitiveness to our tool as a future work.

References

Andersen, L. O. (1994). *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen.

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.

Blanchet, B. (1998). Escape analysis: Correctness proof, implementation and experimental results. In *POPL*, pages 25–37. ACM.

Bond, M. D. and McKinley, K. S. (2007). Probabilistic calling context. In *OOPSLA*, pages 97–112. ACM.

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.

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.

Denning, D. E. and Denning, P. J. (1977). Certification of programs for secure information flow. *Commun. ACM*, 20:504–513.

Ghiya, R. and Hendren, L. J. (1998). Putting pointer analysis to work. In *POPL*, pages 121–133. ACM.

Gough, B. J. (2005). *An Introduction to GCC*. Network Theory Ltd, 1st edition.

Hardekopf, B. and Lin, C. (2007). The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI*, pages 290–299. ACM.

Jovanovic, N., Kruegel, C., and Kirda, E. (2006). Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Symposium on Security and Privacy*, pages 258–263. IEEE.

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

Lattner, C., Lenharth, A., and Adve, V. S. (2007). Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI*, pages 278–289. ACM.

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

Pearce, D. J., Kelly, P. H. J., and Hankin, C. (2004). Efficient field-sensitive pointer analysis for C. In *PASTE*, pages 37–42.

Pereira, F. M. Q. and Berlin, D. (2009). Wave propagation and deep propagation for pointer analysis. In *CGO*, pages 126–135. IEEE.

Pistoia, M., Flynn, R. J., Koved, L., and Sreedhar, V. C. (2005). Interprocedural analysis for privileged code placement and tainted variable detection. In *ECOOP*, pages 362–386.

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.

Rochlis, J. A. and Eichin, M. W. (1989). With microscope and tweezers: the worm from MIT's perspective. *Commun. ACM*, 32:689–698.

Sagiv, M., Reps, T., and Wilhelm, R. (1998). Solving shape-analysis problems in languages with destructive updating. *TOPLAS*, 20(1):1–50.

Sagiv, M., Reps, T., and Wilhelm, R. (2002). Parametric shape analysis via 3-valued logic. *TOPLAS*, 24:217–298.

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.

Steensgaard, B. (1996). Points-to analysis in almost linear time. In *POPL*, pages 32–41.

Stroustrup, B. (2007). Evolving a language in and for the real world: C++ 1991-2006. In *HOPL*, pages 1–59. ACM.

Team, J. (2006). The java HotSpot virtual machine. Technical Report Technical White Paper, Sun Microsystems.

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.