A General Dynamic Information Flow Tracking Framework for Security Applications

Lap Chung Lam, Tzi-cker Chiueh
Rether Networks, Inc.
75 Health Sciences Drive suite 111
Stony Brook, NY 11790, USA
{lclam, chiueh}@rether.com

Abstract

Many software security solutions require accurate tracking of control/data dependencies among information objects in network applications. This paper presents a general dynamic information flow tracking framework (called GIFT) for C programs that allows an application developer to associate application-specific tags with input data, instruments the application to propagate these tags to all the other data that are control/data-dependent on them, and invokes application-specific processing on output data according to their tag values. To use GIFT, an application developer only needs to implement input and output proxy functions to tag input data and to perform tag-dependent processing on output data, respectively. To demonstrate the usefulness of GIFT, we implement a complete GIFT application called Aussum, which allows selective sandboxing of network client applications based on whether their inputs are "tainted" or not. For a set of computation-intensive test applications, the measured elapsed time overhead of GIFT is less than 35%.

1 Introduction

Information flow tracking refers to the ability to track how the result of a program's execution is related, via either data or control dependencies, to its inputs from the network, the file system and any other external parameters such as environment variables and command line arguments. To accurately track information flow, each piece of input should be assigned a tag, which could be a bit (e.g. taint bit) or a pointer to an arbitrarily complex meta-data structure, and for each assignment operation, the tag of the assignment operation's left-hand side is derived from the tags at its right-hand side according to certain tag combination rules. Different information flow tracking applications require different types of tag and use different tag combination rules. Moreover, to accommodate modern network services that consist of multiple programs communicating with each other through messages, it is essential to track information flow not only within a single process, but also across processes, machines, and even Internet sites.

Tracking information flow within a program requires following edges in the program's data flow graph, and propagating tags from the dependees to the dependents. Given an application program and its application-specific tag initialization and propagation rules, in theory a compiler should be able to simulate tag propagation and compute the tag values associated with program variables through abstract interpretation and symbolic execution. In practice, it is not feasible to statically determine the tag value of every program variable because of pointer aliasing and loops with a dynamic bound. Therefore, accurate information flow tracking has to be done dynamically. This requires instrumentation of programs so as to initialize, propagate and combine tags appropriately as their corresponding program variables are being processed. While it is possible for application programmers to take on this instrumentation task, it would be ideal if a compiler could automate the entire program instrumentation process and make it more efficient and less error-prone. This paper describes the design and implementation of such a compiler called GIFT (General dynamic Information Flow Tracking).

GIFT is a compiler for programs written in the C language that takes programmer-specified application-specific rules for tag initialization, propagation and combination, and automatically instruments programs so as to execute these rules as part of the program execution. GIFT not only significantly improves the accuracy of information flow tracking by applying dynamic data/control flow analysis, but also largely automates the process of implementing information flow tracking into individual applications. To the best of our knowledge, GIFT is the first known application-independent implementation framework for information flow tracking, which could be quickly customized through incorporation of application-specific knowledge.

Compared with static information flow tracking [10, 16], GIFT guarantees accurate information flow tracking because it follows data and control dependencies that actually take place at run time. Compared with statistical correlation or machine learning approach to information flow tracking [17], GIFT is completely automatic without requiring a time-consuming and labor-intensive training process. Finally, GIFT makes it possible to in-

strument the same program differently when it is used in different applications or systems, in a way tailored to their security requirement. Therefore, GIFT allows clean decoupling of application logic from tag manipulation logic. In summary, GIFT is an enabling technology that could be immediately applied to a wide variety of information flow accounting applications, including information flow control, intrusion impact assessment, execution trajectory analysis, etc.

To demonstrate the usefulness of the GIFT framework, we build a tool based on GIFT called Aussum, which could automatically instrument arbitrary network applications to track the provenance of information objects and enable *selective sandboxing* on the execution of application programs when they operate on information objects from suspicious sources. Aussum solves a long standing problem for existing behavior blocking or sandboxing systems [1]: How to minimize disruption to legitimate applications while stopping all malicious attacks? By leveraging GIFT's accurate information flow tracking, Aussum allows the same application to be sandboxed in a different way when it operates on different input objects.

2 General Dynamic Information-Flow Tracking Framework (GIFT)

GIFT associates each program variable in an application with a 4-byte tag, which could correspond to a piece of metadata that annotates the variable or a pointer to another data structure that annotates the variable. Being an application-independent information flow tracking framework, the GIFT compiler does not interpret the tags, and leaves their interpretation to the application programmers. Therefore a tag can be used to represent different metadata for different types of information flow tracking, for example, packet ID, user ID, file name, network IP address, security class, etc. The main job of the GIFT compiler is to insert code that calls programmer-provided tag initialization and combining functions at the times specified by the programmers. The GIFT compiler is also responsible for passing tags throughout an entire program.

To use the GIFT compiler, the application programmer needs to specify a set of *interception points*, each of which corresponds to a function in the original application, and a proxy function that should be called instead at each of the interception points. Currently, GIFT supports the following three types of interception points:

- **Input Channel:** These correspond to functions that bring external data into an application's address space, such as `read()` or `write()` functions for the file system, the network, and share memory regions. When these functions are called, the GIFT compiler redirects the calls to their corresponding proxy functions, which perform tag initialization for those program variables that are allocated to receive external inputs.

- **Output Channel:** These correspond to functions that move data in an application's address space to the outside world, i.e., a file system, a remote node, or another process. Output channels are intercepted and redirected in the same way as input channels to their proxy functions, which typically examine tags for the data to be output and make certain decisions. For example, a user can intercept `write` system calls using a proxy function `write_proxy(int fd, void *buf, int side)`, which can examine the tag associated with the variable pointed by `buf` and decide to reject a `write` call that tries to write a password file to a socket descriptor.

- **Assignment Statement:** After every assignment statement in a program, the GIFT compiler inserts a call to a programmer-provided function (by default its name is `gift_set_tag`) to combine the tags associated with program variables at the right-hand side to form the tag associated with the variable at the left-hand side. The proxy function takes the addresses of the tags of all variables in the assignment statement as the arguments, and performs application-specific tag value propagation from the right-hand side to the left-hand side.

In addition, GIFT provides the following library functions for the programmer-provided functions to access the tags of a function call's arguments:

- `void * gift_lookup_tag(void *address)`. If the input parameter is a pointer to a memory block, this function looks up the address of the tag associated with the memory block pointed by `address`.
- `void * gift_lookup_parameter_tag(int index)`. If the input parameter is not an pointer, this function returns the address of the tag associated with the `index`-th argument.
- `void gift_save_return_tag(void *return_address, void *tag)`. If a proxy function needs to return a data value, this function is called to save a copy of its tag into the shadow stack.

GIFT is derived from GCC 3.3.3. To add dynamic information flow tracking to an applicationi using GIFT, the user needs to implement proxy functions for input/output channels and `gift_set_tag` for assignment statements in a object file and link the original program with the object file. For example, if a developer wants to compile a file called `myprogram.c`, she should invoke GCC using "`gcc -fgift -fgproxy=myproxy.pro myprogram.c myproxy.o`". The `-fgift` option enables GIFT's instrumentation. The names of the intercepted functions and their corresponding proxy functions are specified in the file called `myproxy.pro`. The file `myproxy.o` contains the developer-provided proxy functions and the tag propagation function for assignment statements.

3 Design and Implementation of the GIFT Framework

3.1 Tag Management

GIFT associates a tag with each data memory block, which could be a local variable, a global variable, or a

```
1  int buffer[10];
2
3  void work(void)
4  {
5    int a, b, c, d;
6    b = 10;
7    c = 20;
8    read(socket_fd, &a, 4);
9    d = decode(&a, b, c);
10   write(output_fd, &d, 4)
11 }
12
13 int decode(int *a,int b,int c)
14 {
15   int r;
16   r = *a+b+30;
17   return r;
18 }
```

(A) Original

```
1  int buffer[10];
2
3  void work()
4  {
5    int a, b, c, d;
6    int a_tag_info=0, b_tag_info=0, c_tag_info=0, d_tag_info=0;
7    int fun_index;
8    fun_index = gift_add_locals_to_tree(2,
9      &a, 4, &a_tag_info, &d, 4, &d_tag_info);
10
11   b = 10;
12   b_tag_info = 0;
13   c = 20;
14   c_tag_info = 0;
15   gift_save_argument_tag(aussum_read, 1, &socket_fd_tag_info)
16   read_proxy(socket_fd, &a, 4);
17   gift_save_argument_tag(decode,2,&b_tag_info,1,&c_tag_info,2);
18   d = decode(&a, b, c);
19   gift_copy_return_tag(decode_return_address, &d_tag_info, 0);
20   gift_save_argument_tag(aussum_write, 1, &output_fd_tag_info)
21   write_proxy(output_fd, &d, 4)
22   gift_remove_locals_from_tree(fun_index);
23 }
24 int decode(int *a, int b, int c)
25 {
26   int r;
27   int r_tag_info=0, b_tag_info, c_tag_info;
28   gift_init_parameter_tag(decode,2,&b_tag_info,1,&c_tag_info,2);
29
30   r = *a+b+30;
31   gift_set_tag(&r_tag_info, 0, 2,  a, 1, &b_tag_info, 0);
32
33   gift_save_return_tag(return_address, &r_tag_info);
34   return r;
35 }
36
37  __GLOBAL_FileName()
38  {
39    gift_add_globals_to_tree(1, buffer, 40);
40  }
```

(B) Transformed

Figure 1. *This code segment illustrates how GIFT optimizes away splay tree look-ups by directly accessing a memory block's tag, which is stored in a shadow variable, if it is accessed through its name. The statements in the bold font are inserted by GIFT.*

memory area returned by a `malloc()` call. Because a data memory block can be accessed through pointers, it is necessary to provide a mechanism to identify a memory block's tag from a pointer to the memory block. The *explicit look-up* approach [6], uses a separate search data structure to associate a pointer with its metadata, and completely does away with modification to pointer representation. For each data memory block, this scheme creates a node in a *splay tree*, which stores the base address and the size of the memory block, and the memory block's metadata. Given a pointer, this scheme first looks it up in the splay tree to identify the corresponding metadata. A pointer matches a splay tree node if it falls within the node's extent as defined by its base and size.

Originally we implemented GIFT using the splay tree scheme, where each tree node contains the `base, extent, and tag` of a data memory block. However, we found that the splay tree scheme incurs a serious performance penalty for computation-intensive applications due to tree lookups. To reduce this performance penalty, we use a shadow variable to store the tag of every global/local memory block instead of putting the tags into the splay tree. The key idea is that if a memory block is accessed through its name, the GIFT compiler directly looks up its tag in its shadow variable without looking up the splay tree. However, if a memory block is accessed through a pointer, GIFT creates a splay tree node for it, which contains a pointer called `int *tag_ptr` that points to the shadow variable holding the memory block's tag. When a memory block is accessed through a pointer, GIFT looks up the splay tree to locate its corresponding shadow variable. If a memory block is returned by `malloc`, GIFT stores its tag in the `int tag` field of its corresponding splay tree node; in this case the `tag_ptr` pointer of its splay tree node points to its `int tag` field.

Figure 1 illustrates GIFT's instrumentation using a simple program and its transformed version. All variables whose name ends with a `_tag_info` suffix in Figure 1(B) are shadow variables inserted by the GIFT compiler. Line 12 and 14 in Figure 1(B) show that the tags of `b` and `c`, i.e. `b_tag_info` and `c_tag_info`, are directly accessed because these two variables are accessed through names. In addition, the GIFT compiler inserts the following functions into a program to manage tags:

- *int gift_add_locals_to_tree(int number, [void *address, int size, void *tag_addr])*. This function is inserted in each function's prologue to create splay tree nodes for all local data variables and input parameters, whose addresses are assigned to pointers, as illustrated by the call at Line 8 of Figure 1(B). This function takes a variable number of arguments. The first argument `number` indicates how many triples of `[void *address, int size, void *tag_addr]` are passed to the function, where `address` is the base address of a data variable, `size` is the size of the data variable, and `tag_addr` is the address of the associated shadow variable. The return value of this function is used to remove the tree nodes when the function returns.
- *void gift_remove_locals_from_tree(int fun_index)*. This function is inserted in each function's epilogue to remove all tree nodes allocated for the current function. The parameter `fun_index` is the transaction ID returned by `gift_add_locals_to_tree`.

- *void gift_add_globals_to_tree (int number, [void *address, int size, void *tag_addr])*. This function is used to allocate tree nodes for all global and static data variables in a source file. Its prototype is similar to that of `gift_add_locals_to_tree` except that it does not need to return a transaction ID because it is not necessary to explicitly free the tree nodes allocated to the global and static data variables. GIFT creates a global constructor function for each source file as indicated by the function at Line 37 of Figure 1(B). The compiler always generates code to call global constructor functions before the main function is executed.
- *void gift_add_heap_to_tree(void address, int size)* is called in the `malloc` proxy functions to create tree nodes for heap memory blocks allocated by `malloc`.
- *void gift_remove_heap_from_tree(void *address)* is called in the `free` proxy function to remove tree nodes associated with heap memory blocks.

3.2 Dynamic Tag Tracking

For each assignment statement in the program, GIFT inserts a call to `gift_set_tag`, which sets the tag of the left-hand-side memory block based on those of the right-hand-side blocks, as shown in Line 31 of Figure 1(B), which corresponds to the assignment statement in Line 16 of Figure 1(A). In this case, the address of `r`'s shadow variable, the address of the memory block pointed to by `a`, and the address of `b`'s shadow variable are passed to `gift_set_tag`, which only needs to look up the splay tree for the address of `a`'s shadow variable. Because GIFT does not understand how a tag is used, after the addresses of all the shadow variables involved in an assignment statement are resolved, `gift_set_tag` calls the user-supplied function `gift_do_set_tag(void *ltag, int number, void *tags)` to actually propagate the tags. The argument "`void *tags`" of this function are the addresses of the shadow variables of the right-hand-side memory blocks.

When an input argument of a function call is not a pointer, GIFT needs to propagate its tag to the callee. Instead of passing the tag through the standard stack, which could cause compatibility problems with legacy code, GIFT allocates a shadow stack, one per thread, to pass the tags of non-pointer function call arguments. Specifically, the caller calls `gift_save_argument_tag`, e.g., Line 17 of Figure 1(B), to place on the shadow stack the entry point of the callee and the tags of non-pointer arguments before the call. The callee calls `gift_init_parameter_tag` to retrieve the tag for each parameter. `Gift_init_parameter_tag` first compares the callee's entry point with the entry point stored on the shadow stack. If they match, the tags of the parameters of the callee are initialized with the tag values passed through the shadow stack. If two entry points do not match, this indicates that callee is called from a legacy function, which does not push the tags of the actual arguments to the shadow stack. In this case, `gift_init_parameter_tag` initializes each parameter tag with zero.

If a function returns a non-pointer value back to the caller, GIFT also needs to propagate the tag of the return value to the caller. It uses the same shadow stack mechanism as in the case of a function call. But the arguments pushed to the shadow stack are the return address and the address of the return value's tag, as shown by the call to `gift_save_return_tag` at Line 33 of Figure 1(B). The first argument `return_address` is the return address of the callee, which is generated by the GIFT compiler. The caller compares the return address on the shadow stack with the call site, and propagates the tag of the return value accordingly if they match by calling `gift_copy_return_tag`, as shown in Line 19 of Figure 1(B). The first argument `decode_return_address` of the call to `gift_copy_return_tag` is the call site/return address of the call to the function `decode` at Line 18. `Gift_copy_return_tag` compares this return address with the return address stored in the shadow stack. If the two return addresses do not mach, the function call is returned from a legacy function, which does not place the return tag on the shadow stack. In this scenario, `gift_copy_return_tag` treats the return tag as zero.

When a GIFT function calls a legacy function, the tags that the GIFT function puts on the call shadow stack is ignored. When the GIFT function returns, the information that the GIFT function puts on the return shadow stack is ignored by the legacy function. In either case, the program continues working without any disruption. Consequently, the above tag propagation scheme solves the compatibility problem associated with the shadow variable approach.

To avoid complete re-compilation of the LIBC library, GIFT includes a set of wrapper functions for memory and string copying library functions in LIBC, such as `memcpy` and `strcpy`. The GIFT compiler redirects all calls to memory/string copying functions to their corresponding wrapper functions, which properly set the tag of the destination memory block using the tags associated with the source memory blocks.

3.3 Array, Union, and Structure

GIFT treats each array, union, and structure variable as a single memory block, and allocates to it a single tag. This means that the tag of an array/union/structure should contain the most important metadata associated with that variable. If fine-grained tagging, for example one tag for each field in a structure variable, is needed, the developer can use the structure variable's shadow variable as a pointer to a more complex tag data structure, which can contain many sub-tags.

3.4 Slicing Optimization

Blindly intercepting all assignment statements and function calls/returns in a program may result in many unnecessary calls to GIFT library functions when only parts of the program involve data read from input channels. To focus only on those program statements that

manipulate data related to input channels, GIFT uses the program slicing result from a commercial tool called Codesurfer [5]. Given a set of input and output functions that users want to intercept, GIFT computes a forward slice of the original program from the input functions and a backward slice from the output functions, and takes the intersection between the two slices. The resulting program segment is a set of statements that GIFT instruments because they are affected by input data and their results may be used by output functions.

While conceptually promising, the performance gain from this optimization depends on the pointer usage in the applications. If an application uses pointers and/or function pointers extensively, this slicing optimization does not help much, as Codesurfer's pointer analysis is not very effective. Even when Codesurfer just performs flow-insensitive pointer analysis, it requires an inordinate amount of memory resource and could run very slowly when one turns on the most accurate pointer analysis option. In the case of less accurate pointer analysis option, the result that Codesurfer produces is not much different from the original program, and as a result slicing does not result in any noticeable performance improvement.

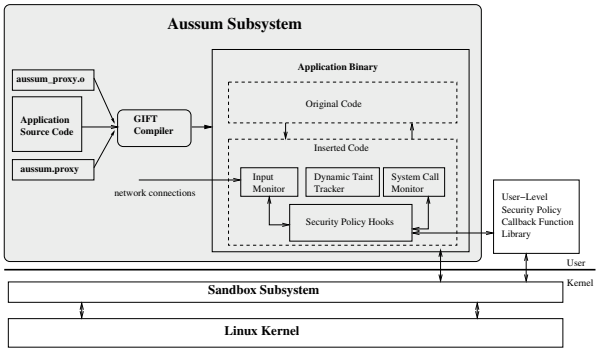

Figure 2. *The Aussum compiler is built from the GIFT framework. The Input Monitor intercepts data read from a network connection and marks it according to a configurable security policy. The Dynamic Taint Tracker propagates the mark throughout the program. Finally the System Call Monitor marks files and system call arguments that are derived from network inputs.*

4 Selective Application Sandboxing Using GIFT

4.1 Overview

Many end user computers are infected by malicious programs because the users knowingly or unknowingly download from the network objects containing malicious programs, by reading email attachments, file transfer, web browsing, and peer-to-peer file sharing. Most existing anti-malware tools are based on signatures and therefore cannot protect end user machines from zero-day attacks. Behavior blocking or sandboxing [1], which monitors and restricts network applications' execution according to a pre-defined security policy, is considered a better alternative against zero-day attacks, because it focuses on benign program behaviors rather than malicious ones. However, in practice, the behavior blocking technology exhibits two pitfalls. First, behavior blocking could disrupt the operation of legitimate applications because, for safety, the sandboxing policy is typically set stricter than necessary. One way to solve this problem is to apply program analysis techniques to extract highly accurate sandboxing policies directly from application programs' source or binary code [7].

Second, existing behavior blocking systems do not support *selective sandboxing*, which sandboxes the same application differently depending on the mode in which the application is currently in. To solve this problem, some existing sandboxing systems such as Tiny Firewall [18] require the end user to specify both the applications that need to be sandboxed as well as the policies to be used. This approach is inconvenient and error-prone. Eventually the user is likely to choose convenience over security and turns off sandboxing completely. Other systems such as SEES [8] attempt to address this problem by transparently intercepting the file download path of web browsers and email clients, and marking each downloaded file. These systems then sandbox the execution of a program if it itself is marked or the object it operates on is marked (such as Mcrosoft WORD). However, this approach is limited because the way they mark downloaded files is very specific to individual applications and therefore cannot be generalized to arbitrary network applications.

This section presents the design and implementation of Aussum, which is an application of the GIFT compiler framework that can automatically instrument network applications so as to enable selective sandboxing. Aussum leverages GIFT's dynamic information flow tracking capability to automatically mark contents downloaded from the Internet. When marked contents are written to a file, Aussum marks the file in such a way that any subsequent execution using that file is sandboxed. In addition, If a network application uses marked contents as input arguments to sensitive system calls such as `exec`, `open`, and `unlink`, these system call invocations are also sandboxed. Because of the fine-gained information flow tracking capability, Aussum allows a legitimate application such as Microsoft's IE or WORD to run with full privilege when it operates on local files, but is properly sandboxed when it operates on objects downloaded from the network. Moreover, Aussum injects this selective sandboxing capability to network applications in a way that is completely transparent to the applications' developers and users.

Figure 2 shows the system architecture of Aussum, which adds three modules into a network application. The *Input Monitor* marks as *tainted* input packets from network connections that are considered suspicious according to a security policy stored in the *Security Policy Hooks module*. The *Dynamic Taint Tracker* propagates the taint attribute of data items across a program's

computation. The *System Call Monitor* checks the arguments of sensitive system calls such as `unlink`, `open`, and `exec`, and invokes the operating system's sandboxing mechanism when their arguments are tainted. In addition, when tainted data is written to a file, the System Call Monitor marks the file as tainted according to the same security policy. The security policy in the *Security Policy Hooks module* is represented in the form of callback functions, which determine if a network connection should be considered suspicious and/or if a file should be marked as tainted. When a tainted file is executed or used as input to a new process, Aussum automatically sandboxes the corresponding process. Aussum can work with any existing sandbox system as long as the sandbox system implements the security policy hook functions required by Aussum.

```
1  Struct au_descriptor_t{
2    int suspicious;  /* tag to identify network descriptor */
3    int type;        /* file or socket */
4    char *name;      /* ip or file name */
5  } *au_descriptor[max_open_files];
6
7  int aussum_connect(int s, struct sockaddr *addr, socklen_t *addrlen)
8  {
9     int ret;
10    ret = connect(s, addr, addrlen); /* call the original function */
11    if(ret != -1){                   /* connection successes */
12       . . .                  /* create descriptor entry and set the ip
         /* callback function implemented by the sandboxing system*/
13       if(mark_connection_ptr != NULL){
14         if(mark_connection_ptr(addr)) /*let a sandboxing system to ma
15           au_descriptor[s]->suspicious = TRUE;
16         else
17           au_descriptor[s]->suspicious = FALSE;
18       }
19       else
20         au_descriptr[s]->suspicious = TRUE; /* default operation */
21    }
22    return ret;
23 }
```

Figure 3. *Proxy Function for* `connect`.

4.2 Input Monitor

Because both network connections and files are accessed through descriptors using functions such as `read` and `fread`, the Input Monitor needs to first identify tainted descriptors so that it could properly mark results of `read` and `fread` calls as tainted. In each application, Aussum maintains a descriptor table to store information about each opened file or socket descriptor. To mark a descriptor, Aussum uses GIFT to redirect each call to `accept`, `connect`, `dup`, or `dup2` to its corresponding proxy function, which creates a descriptor table entry for each new descriptor and marks it according to the security policy. More specifically, if a security policy callback function `int mark_connection(sockaddr *addr)` exists, these proxy functions call this function to determine how to mark a new descriptor; otherwise, they simply mark all socked descriptors as tainted. For example, the proxy function for `connect` works as in figure 3.

Line 10 of the proxy function in figure 3 calls the original `connect` function, and Line 13-17 consults with the underlying sandboxing system to determine if a network connection is suspicious. Through this callback function mechanism, the underlying sandboxing system can implement its own security policy. For example, all data from a local network may be considered as safe. If the variable `mark_connection_ptr`, which points to the callback function `int mark_connection(sockaddr *addr)`, is null, the underlying sandboxing system does not have any specific security policy, and Aussum simply marks every network connection descriptor as tainted. The parameter `addr` gives the sandboxing system an opportunity to examine the remote IP address to decide if the connection should be marked tainted.

In Aussum, a tag's value is either 0 or 1, where 1 means tainted and 0 means not tainted. To initialize the tag for data read from a descriptor, the Input Monitor intercepts such system calls and LIBC function calls as `read`, `fread`, `fgets`, `recv`, and `recvfrom` using proxy functions, each of which first calls the original function, and then checks if the socket/file descriptor is marked as tainted by looking up the descriptor table. If data is read from a tainted descriptor, Aussum locates the corresponding node in the splay tree, and sets its tag to 1; otherwise the tag is set to 0.

4.3 Dynamic Taint Tracker

GIFT requires developers to implement their own `gift_do_set_tag` to propagate tag values across assignment statements. In Aussum, the tag of the left-hand-side memory block of an assignment statement is the bitwise OR of the tags of the memory blocks at its right-hand side. That is, if any memory block on the right-hand side is tainted, the left-hand-side memory block is also tainted. The following code shows the Aussum's implementation of `gift_do_set_tag`:

```
void gift_do_set_tag(void *lhs_tag, int num,
                     void *rhs_tags)
{
  int i, tmp = 0;
  int **rt = (int **)rhs_tags;
  for(i = 0; i < num; i++)
    tmp =  tmp|(*rt[i]);
  *(int *)lhs_tag = tmp;
}
```

4.4 System Call Monitor

The System Call Monitor marks as tainted output files that contain tainted data. If an executable file is tainted, the sandboxing system will sandbox its execution when it is invoked. If a document is tainted, the sandboxing system will sandbox the application that opens it. This mechanism can effectively thwart MS WORD VB Macro virus attacks and recent Windows WMF attacks. Ideally the taint attribute of a file should be an inherent part of the file so that when it is copied to a new file, its taint attribute is copied automatically. Under an UNIX-like operating system, there is no unused file attribute that can be used for this purpose. One possible way to solve this problem is to create a group called TAINTED. If a file is considered tainted, its group owner attribute is set to TAINTED. However, Aussum leaves this decision to the underlying sandboxing system. That is, Aussum provides a callback function for the underlying sandboxing system to specify how files should be marked as tainted. To support file marking, Aussum intercepts file output functions such as `write`

and `fwrite` using proxy functions. For example, the proxy function for `fwrite` is shown in figure 4.

```
1 size_t aussum_fwrite(const void *ptr, size_t size, size_t nitems, FILE *str
2 {
3   int ret;
4   int fd;
5
6   ret = fwrite(ptr, size, nitems, stream);/*call the original function*/
7   if(ret == nitems){
8     if(aussum_mark_file_ptr != NULL){ /* hook function for marking files*/
9       fd = fileno(stream);
10      if(!au_descriptor[fd]->marked){ /* we have not marked the file */
11        if(is_suspicious(ptr))   /* is the memory location suspicous? */
12          asussum_mark_file_ptr(fd, au_descriptor[fd]->name);
13        au_descriptor[fd]->Marked = 1;
14      }
15    }
16  }
17  return ret;
18 }
```

Figure 4. *Proxy Function for* `fwrite`.

Line 8 of figure 4 checks if the underlying sandboxing system provides a file marking function. Through the function pointer `aussum_mark_file_ptr`, the underlying sandboxing system can make its own decision on whether a file should be marked as tainted and if so how. For example, even if data written to a file are marked as tainted, the underlying sandboxing system can choose to mark the resulting file as not tainted because the file type is `.txt`.

The embedded scripts present a technical challenge to Aussum because it cannot afford to sandbox at all time the application that downloads them, such as IE or Firefox. Ideally, one should sandbox the downloading application only when it is executing embedded scripts, but leave it alone to run at full privilege when it operates on local files. Aussum solves this problem by selectively turning on and off sandboxing for a network application based on whether the operands it is currently operating on is tainted or not. It makes the following three assumptions. First, a malicious script cannot cause damage if it cannot make system calls. Second, a malicious script can only make system calls through the interpreter that interprets it. Finally, if a malicious script causes damage through system calls, some of the system call arguments must be tainted. Based on these assumptions, Aussum intercepts sensitive system calls such as `open` and `exec` to check if any of the input arguments is tainted. By default, Aussum terminates any applications that attempt to invoke a sensitive system call using tainted arguments, except those that use file names read from the network to open or create non-existing files. Aussum allows applications to open the non-existing files because applications such as Firefox and IE need to save to cache or temp directory files whose names are derived from an html document file. Although this default policy may open doors to deny of service attack, at least it prevents malicious scripts from modifying or deleting the existing files. This policy is also consistent with the taint data policy used in Perl.

Unfortunately, this policy may break certain peer-to-peer applications, which allow a remote application to request a local application to perform sensitive system calls. To address this issue, Aussum again provides a callback function called `aussum_check_argument` to allow the underlying sandboxing system to override the default policy by allowing, for example, read and write accesses to certain directories or execution of certain existing binaries.

4.5 Callback Functions as Security Policy

Aussum's Security Policy module provides the following three callback functions for the underlying sandboxing system to determine when to treat a network connection suspicious, when to mark a file tainted, and when to reject a sensitive system call invocation when it uses tainted arguments:

```
int aussum_mark_connection(sockaddr *addr);
/*if this function returns true, the connection to
  addr should be marked as suspicious.*/

int aussum_mark_file(int descriptor, char *file_name);
/*this function marks the output file as tainted.*/

int aussum_check_argument(char *function_name,
                          char *argument);
/*if the return value is 0, terminate the application.
  if the return value is 1, simply return without
  makeing the system call.
  if the return value is 2, make the system call.*/
```

Since these three functions are very simple, it takes minimal effort to add these functions into an existing sandboxing system.

4.5.1 Performance Analysis

4.6 Performance Evaluation

4.6.1 Methodology

Because Aussum is designed to provide selective sandboxing for applications running on desktop machines, we chose a set of network client applications listed in table 1 to evaluate the performance overhead of Aussum's taint attribute propagation mechanism. Because every assignment statement in a network application needs to be instrumented, we purposely chose CG, WGET, and LYNX, which are computation-intensive, to stress-test the efficiency of the GIFT compiler. CG is a newsgroup binary downloader, which needs to parse every mail in a group, download and decode all the attachments. WGET parses every downloaded html file to find new files to download. LYNX parses and displays a web page on a terminal. We modified LYNX to exit immediate after the page is displayed, so that we could measure the performance of LYNX programmatically. For the interactive programs such as CG, we also made similar modifications to them so that they could perform the required operations and exit immediately.

The server machine used in this study is a 2.8GHz P4 machine with 256MB RAM, and it runs Fedora core 3.0. The client machine is a 2.8GHz P4 machine with 1.5GB RAM that runs Redhat 7.3. To test the ftp client, we fetched a 6MB file. To test the CG program, we set up a newsgroup with a 40KB image and two 700KB binary programs, and used CG to read all the information from the newsgroup and download the image and the binaries. To test Gnut, which is a peer-to-peer application, we fetched a 500KB file and a 10KB file. We used WGET to download Apache's user manual files from an Apache server, and used LYNX to download and display the main page of the Apache manual.

The Aussum prototype includes a simple sandboxing system, which provides callback functions to mark

Application	Description	Lines of code	Space Overhead			
			gcc (bytes)	splay	shadow/splay	shadow/splay/slice
tnftp	FTP client	35,573	183,820	64.11%	61.49%	50.71%
yafc	FTP client	26,587	204,444	49.93%	44.14%	34.33%
gnut	Gnutella client	22,298	190,804	58.11%	46.44%	31.65%
cg	newsgroup binary downloader	10,974	54,260	79.28%	76.81%	61.21%
wget	mirroring tool	35,018	164,904	50.15%	48.54%	43.57%
lynx	text web browser	161,605	1,038,340	36.12%	29.81%	28.85%

Table 1. *Description of the test applications used in the performance evaluation study and the increase in binary size for each of three versions of Aussum.*

Application	Splay Tree		Shadow/Splay		Shadow/Splay/Slice	
	Elapsed Time Overhead	CPU Time Overhead	Elapsed Time Overhead	CPU Time Overhead	Elapsed Time Overhead	CPU Time Overhead
tnftp	6.52%	24.14%	0.70%	19.54%	0.90%	16.54%
yafc	6.41%	78.18%	2.92%	15.45%	1.32%	10.00%
gnut	89.12%	257.14%	5.44%	28.57%	1.16%	20.00%
cg	308.97%	1120%	8.97%	180%	6.41%	160.00%
wget	77.06%	534.76%	14.29%	127.2%	9.75%	117.13%
lynx	152.86%	400%	30.00%	166.67%	31.43%	166.67%

Table 2. *Performance overheads of the three Aussum versions when compared with vanilla GCC.*

tainted data and files and is organized as a loadable Linux module. When an application opens an existing file for write by invoking `fopen` or `open` with tainted input arguments, the sandboxing system copies the file to some temporary directory and opens it there, in a way similar to FVM [20] and Alcatraz [9]. To evaluate the effectiveness of GIFT's tag propagation mechanisms, we tested three versions of Aussum: Aussum using the pure splay tree scheme (`splay`), Aussum using the combined splay tree/ shadow variable scheme (`shadow/splay`), and Aussum using the combined splay tree/shadow variable scheme with slicing optimization (`shadow/splay/slice`).

We measured Aussum's performance overhead using both elapsed time and CPU time. Because the elapsed time includes disk I/O time, the elapsed time overhead could be much smaller than the CPU time overhead. Because Aussum targets at network client applications, the elapsed time overhead is a more appropriate measure because it reflects the user's perception of the slowdown due to GIFT's instrumentation.

Table 2 shows that the CPU time overhead of Aussum ranges from 24.14% to 1120% for the pure splay tree scheme for tag representation. As expected, the pure splay tree scheme incurs the highest performance penalty for computation-intensive programs. For example, the run-time overhead of CG is 1120% (or 12 times as slow), and this high overhead is attributed to the decode computation in CG, where taint attributes are propagated across each decoding computation step. The performance overhead for WGET and LYNX is also very high (534.76% and 400% respectively), because they both need to parse html files. In contrast, the performance overhead of the pure splay tree scheme for the ftp client program is relatively small, because it just fetches files and writes them to disk without performing any computation.

The combined shadow variable/spray tree tag management scheme is very effective. The CPU time overhead ranges from 15.45% to 180%. In particular, the CPU time overhead for CG drops from 1120% to 180% as it changes the tag representation from the pure splay tree scheme to the combined scheme. Table 3, which lists the number of splay tree look-up in each run, demonstrates that the reason why the combined scheme is so effective is because it eliminates most of the splay tree look-ups. More concretely, under the pure splay tree scheme, CG needs to look up the splay tree 2,975,687 times, but only 100,178 times under the combined splay tree/shadow variable scheme.

Application	Splay Tree	Shadow/Splay	Shadow/Splay/Slice
tnftp	23,032	4,879	4,751
yafc	51,369	14,699	14,433
gnut	72,136	24,265	20,796
cg	2,975,687	100,178	83,138
wget	55,537,361	22,285,332	22,180,508
lynx	193,818	47,894	46,392

Table 3. *Number of tree lookups of each configuration.*

The elapsed time overhead of Aussum's taint attribute propagation when compared with vanilla GCC ranges from 6.41% to 308.97% for the `splay` scheme, from 0.70% to 30.00% for the `shadow/splay` scheme, and from 0.90% to 31.43% for the `shadow/splay/slice` scheme. The worst case occurs with LYNX, which incurs a 30% elapsed time overhead and a 166.67% CPU time overhead. LYNX reads an html file from a network connection and displays it on the screen. Even though the tag propagation code inserted by GIFT can a high CPU time overhead, this overhead is not necessarily visible to the user because the network I/O time and screen output time largely dominate the CPU time.

GIFT applies program slicing in the hope to eliminate unnecessary instrumentation and reduce the performance overhead of tag propagation. This slicing optimization works for some programs but not the others such as TNFTP and LYNX. The main reason is

that Codesurfer's pointer analysis is not very effective against TNFTP and LYNX. When we applied the most accurate pointer analysis option, Codesurfer ran out the 1.5GB memory quickly. Therefore, we had no choice but to use the less accurate pointer analysis option. However, in this case, Codesurfer's slicing result is almost the same as the original program. From Table 3, although program slicing slightly decreases the number of splay tree look-ups for TNFTP and LYNX, the performance overhead of TNFTP and LYNX is actually higher when the slicing optimization is turned on.

4.6.2 Effectiveness of Aussum

We wrote a simple sandbox tool to test the effectiveness of Aussum. The sandbox tool specifies a policy that applications cannot open an existing file with a file name that is obtained from a suspicious host, e.g., any node not on the intranet. We then invoke Aussum-instrumented versions of `wget` and `lynx` to download files from the Internet. Aussum could successfully stop their `open` system call when a file name is read from a non-intranet host and a local file with the same name exists. We alo tested Aussum on several network applications used at end user machines. In all cases, Aussum is able to correctly mark as tainted those files that are downloaded from suspicious network addresses as specified in the security policy. If a file is downloaded from a trusted host or it is generated by the test applications themselves, Aussum marks the file as non-tainted. These experiments show that the taint attribute propagation mechanism of Aussum, which is built on GIFT, can indeed correctly propagate tags from input data to output data.

5 Related Work

Flow Caml [15, 16] is an extension of the Objective Caml language with a type system tracing information flow. Each variable in a Flow Caml program is annotated with a *principal*, which can be any entity such as a user, a file, `stdin`, and `stdout`. Information owned by one principal cannot flow to another principal unless the programmer specifically permits the operation using the keyword `flow`. The example `flow !bob < !alice;;` means that the programmer specifically allows Bob to send information to Alice. To correctly implement a program in Flow Caml, a programmer must first list all principals appeared in the program and carefully layout who can read whose information.

Jif [10, 11] is a java extension, which enables programmers to annotate variables with security labels, as in the declaration `int{o1: r1, r2; o2: r2, r3} x;`. The security policy in this declaration specifies `x` is own by `o1` and `o2`. `O1` allows r1 and r2 to read the data, and `o2` allows `r2`, and `r3` to read the data. In this example, only `r2` can read the data `x` since both owners grant the permission to it. Jif's type checking system verifies that no information from one variable can flow to another if the policies prohibit the transaction. Unlike Flow Caml, which only uses static analysis, security label in Jif can be first class value, which can be assigned and checked at run time.

The weakness of Flow Caml and Jif is that they require programmers to layout the information flow security policies at programming time. Although Jif enables users to set the labels of some variables at runtime, the labels of most of the variables are set at the programming time. Therefore, unlike our GIFT mechanism, Flow Caml and Jif have to modify existing applications to use their information control mechanism. Furthermore, GIFT can be used for the purposes other than security, such as logging application file system and database access requests and operations to enable fast intrusion recovery.

Fenton's Data Mark Machine [4] is an early abstract machine that implements dynamic information flow control. The data mark machine associates each variable and the program counter (PC) with a security class. The value of a variable must be computed from the variable with the same security class. In contrast, GIFT is a practical system that can enable any existing C programs to perform dynamic information flow control or tracking.

Recently, dynamic information flow technique is mainly use to detect control hijacking attacks, such as in the implementations of [17, 2, 13]. The dynamic information tracking system implemented by Sub et al. [17] is a hardware implementation. Every memory byte and register has a one-bit hardware tag to tag the data. All tags are initialized to zero and the operating system tags the data with one if they are from a potentially malicious input channel. Instruction sets are augmented to propagate the tags. The processor ensures that no tagged data can be used as execution control transfer. Minos [2] is also a hardware implementation that is similar to [17].

Newsome and Song [13] implemented the similar mechanism in software. They implemented their TaintCheck system using Valgrind [12]. Valgrind is an x86 emulator that can instrument a program as it is run. Each byte of memory, including the registers, stack, heap, etc., has a four-byte shadow memory to store a pointer to a Taint data structure. Input data that come from untrusted source are marked as tainted, and the TaintCheck system instruments a program at runtime to propagate the taint. All control transfer instructions are intercepted to make sure no tainted data can be used as the transfer destination. The disadvantage of this scheme is its performance overhead, which can be more than 20 times slower. Furthermore, this scheme can use 4 times more memory in the worse case.

Sekar et al. [19] also developed a comprehensive taint analysis system to detect and prevent a wide range of attacks, such as buffer overflow, format string, cross-site scripting, and SQL injection attacks. The way they instrument applications is similar to GIFT except that they use one bit to tag each memory byte, and GIFT use a four-byte shadow variable to tag each data variable. Their system focuses on detecting different attacks while GIFT focuses on providing a set of APIs to allow users to propagate program information in their own way. Therefore, a 1-bit tag is not suitable for GIFT. The 4-byte tag of GIFT can be used as a pointer to point to a more complicate data structure, which can record more information, such as user ID, IP address, and packet ID.

Perl also implements taint check [14] to lock out the security bugs existed in perl scripts. When the taint check is turned on, Perl marks all user input data as tainted. The tainted data may not be used in a call to the functions such as `eval` and `open`. If the interpreter detects an operation that uses tainted data in a manner it considers unsafe, it stops the execution with an error.

Unlike the traditional information flow control mechanisms, which control information flow at application variable level, the Asbestos Operating System [3] implements information flow control at process level. GIFT can also be used to implement information flow tracking between processes. For example, we can use GIFT to compile apache server to intercept `setenv`, which is used to send information to a CGI program, and compile the CGI program to intercept `getenv`, which is used to get information from apache. Then we can implement the information flow tracking in the proxy functions of `setenv` and `getenv`.

6 Conclusion

This paper presents the design and implementation of a general dynamic information flow tracking framework for C programs called GIFT, and a complete application of GIFT called Aussum. GIFT provides an interface for developers to specify their own tag initialization, propagation, and processing functions. Then the GIFT compiler automatically propagates these tags through an application program along data dependencies and control dependencies that actually take place at run time. The flexibility of GIFT allows it to support a wide range of applications. Aussum is an example GIFT application that is designed to minimize the probability that sandboxing or behavior blocking systems disrupt legitimate applications by turning on sandboxing only when absolutely necessary. Although Aussum minimizes the disruption of the sandboxing technology through fine-grained information flow tracking, it incurs a modest performance overhead. Measurements on the first Aussum prototype show that the CPU overhead is less than 170% for computation-intensive applications, and is less than 20% for non-computation intensive applications. The elapsed time overhead is less than 35% even for the computation-intensive applications.

References

[1] A. Conry-Murray. Product focus: Behavior-blocking stops unknown malicious code. *http://www.networkmagazine.com/shared/article/showArticle.jhtml?articleId=8703363&classroom=*, June 2002.

[2] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *37th Annual International Symposium on Microarchitecture*, pages 221–232, December 2004.

[3] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazires, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 17–30, October 2005.

[4] J. S. Fenton. Memoryless subsystems. *Computing Journal*, 17(2):143–147, May 1974.

[5] GrammaTech, Inc. Codesurfer. *http://www.grammatech.com/products/codesurfer/*.

[6] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Proceedings of Automated and Algorithmic Debugging Workshop*, pages 13–26, 1997.

[7] L. C. Lam and T. Chiueh. Automatic extraction of highly accurate application-specific sandboxing policy. In *Seventh International Symposium on Recent Advances in Intrusion Detection*, Sophia Antipolis, French Riviera, France, September 15-17 2004.

[8] L. C. Lam, Y. Yang, and T. cker Chiueh. Secure mobile code execution service. In *Proceedings of USENIX Large Installation Systems Administration (LISA) Conference*, December 2006.

[9] Z. Liang, V. Venkatakrishnan, and R. Sekar. Isolated program execution: An application transparent approach for executing untrusted programs. *19th Annual Computer Security Applications Conference*, December 8-12 2003.

[10] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[11] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Symposium on Operating Systems Principles*, pages 129–142, 1997.

[12] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Proceedings of the Third Workshop on Runtime Verification (RV'o3)*, July 2003.

[13] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS 2005)*, February 2005.

[14] Perl. Perlsec. *http://perldoc.perl.org/perlsec.html*.

[15] F. Pottier and V. Simonet. Information flow inference for ML. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 319–330, Portland, Oregon, Jan. 2002.

[16] V. Simonet. Flow Caml in a nutshell. In G. Hutton, editor, *Proceedings of the first APPSEM-II workshop*, pages 152–165, Nottingham, United Kingdom, March 2003.

[17] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, October 2004.

[18] Tiny Software. Tiny firewall 6. *http://www.tinysoftware.com/home/tiny2?s=780713668641115504 9A1&&pg=content05&an=tf6_home*.

[19] W. Xu, S. Bhatkar, , and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *15th USENIX Security Symposium*, pages 121–136, August 2006.

[20] Y. Yu, F. Guo, S. Nanda, L. C. Lam, and T. Chiueh. A feather-weight virtual machine for windows applications. In *Proceedings of the 2nd ACM/USENIX Conference on Virtual Execution Environments (VEE'06)*, June 2006.