Preventing Format-String Attacks via Automatic and Efficient Dynamic Checking

Michael F. Ringenburg and Dan Grossman
Department of Computer Science and Engineering
University of Washington
Box 352350
Seattle, WA 98195
miker@cs.washington.edu, djg@cs.washington.edu

ABSTRACT

We propose preventing format-string attacks with a combination of static dataflow analysis and dynamic white-lists of safe address ranges. The dynamic nature of our white-lists provides the flexibility necessary to encode a very precise security policy—namely, that `%n`-specifiers in `printf`-style functions should modify a memory location x only if the programmer explicitly passes a pointer to x. Our static dataflow analysis and source transformations let us automatically maintain and check the white-list without any programmer effort—they merely need to change the Makefile. Our analysis also detects pointers passed to `vprintf`-style functions through (possibly multiple layers of) wrapper functions. Our results establish that our approach provides better protection than previous work and incurs little performance overhead.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Invasive software (e.g., viruses, worms, Trojan horses); D.2.0 [**General**]: Protection mechanisms; D.2.4 [**Software/Program Verification**]: Reliability

General Terms

Security

Keywords

Format-String Attacks, Static Analysis, White-Lists, Dynamic Checking

1. INTRODUCTION

The well-known vulnerabilities of systems implemented in the C programming language are unsurprising when we consider C programming from a security perspective:

- C's lack of memory safety essentially means *any* piece of code might modify *any* part of the address space. Moreover, for performance reasons, libraries typically do not check function arguments.
- The *principle of least privilege* [29], probably the closest thing there is to an axiom of security, states that no entity should be given more rights than necessary to complete its task.

The inherent conflict between the two points above is obvious and grows worse as we build ever larger systems in C: Most code is permitted to do much more than it should, particularly with respect to modifying memory. At a high-level, format-string attacks, as well as many other standard exploits, take advantage of this security weakness.

Previous proposals for addressing format-string attacks (and other vulnerabilities in C) include: software-fault isolation or virtual execution [37, 32, 18], hardened libraries [36], run-time detection of illegal writes [7, 23, 13], type-safe dialects or implementations of C [1, 5, 14], static lint-like code analysis for likely errors [15, 33, 2, 10], more sophisticated static analysis [31], and code rewriting techniques [6]. As Section 5 discusses in detail, these projects are all valuable: They catch real bugs and should be used more than they are. However, they tend either to restrict code (e.g., banning `%n` in non-static format strings), or to miss large classes of vulnerabilities (e.g., in wrapper functions that call `vprintf`).

In this paper, we propose a new approach for preventing format-string attacks. We combine the precision and security of run-time approaches with the ease-of-use of static analyses and automatic source transformations. Specifically, at run-time we maintain a dynamically updated white-list of `%n`-writable address ranges. This allows us to encode a very precise security policy—namely, that `%n`-specifiers in format strings should be allowed to modify a memory location *if and only if* the programmer explicitly passes a pointer to it. We use static dataflow analysis to determine automatically which addresses should be in the white-list at any given time. Our source-to-source transformation then uses the knowledge gleaned from static analysis to insert the code that maintains and checks the white-list. Thus the programmer merely needs to update the Makefile and recompile. We have tested our tool on a number of programs, and did not have to change any source files. All the programs ran correctly, and all the format-string vulnerabilities disappeared. Our tool is available for download from our website [26].

CCS'05, November 7–11, 2005, Alexandria, Virginia, USA.

1.1 Format-String Attacks

Since their discovery roughly five years ago [35], format-string attacks have become all-too-common [6]. For example, a recent search on securityfocus.com revealed 62 separate vulnerabilities posted in 2004 that contained the phrase "format string". This has occurred even though (or perhaps because) such vulnerabilities are well-understood [22, 38] and there exist (partial) techniques for avoiding them [6, 31].

The essence of the vulnerability is straightforward:

- User-supplied input is frequently used as the format-string argument to a printing function (such as `sprintf`, `snprintf`, `fprintf`, `vprintf`, `vsprintf`, `vsnprintf`, `vfprintf`, `syslog`, or `vsyslog`).
- The format-string argument to printing functions causes memory writes if the `%n` format specifier appears. Specifically, `%n` writes the number of bytes output by the printing function (prior to the `%n`) to the memory location specified by the corresponding argument.
- Printing functions do not check the number or types of their variable arguments.[1]

Thus, an attacker can perform unauthorized writes by inserting unexpected `%n`'s into user-supplied input strings (e.g., command-line arguments) that eventually get passed to printing functions. For example, if a malicious user calls the following program (adapted from [22]) with the command line argument `"aaaabbbccc%n"`, the value 10 will be written to the address `0x61616161`:

```
int main(int argc, char **argv) {
  char buf[100];
  snprintf(buf, 100, argv[1]);
}
```

The `snprintf` statement first writes "aaaabbbccc" to `buf`. It then pulls the next argument off of the stack, assumes it is an integer pointer, and writes the number of bytes output so far (10 in this example) to the pointed to location. In this case, however, there is no next argument. Thus the first four bytes of `buf` are grabbed off the stack instead, and 10 is written to `0x61616161` (`0x61` is 'a' in ASCII). Furthermore, `%n` ignores the length argument of `snprintf`, and instead writes the number of bytes that it believes *would* have been printed had there been no length restrictions. Thus, this style of attack can be used to write an arbitrary value to an arbitrary location in memory.

Format-string attacks are also possible with specifiers (such as `%s`) that read the memory pointed to by the corresponding argument. However, `%n`-based format-string attacks are the most dangerous, because they allow the attacker to write arbitrary values to arbitrary memory locations (and thus to potentially execute arbitrary code on the victim machine). In contrast, format-string attacks based on other specifiers can only read data, or potentially crash the program (e.g., by following an invalid pointer). A quick scan of format-string attacks in the wild (e.g., [28, 35, 30, 25, 24, 19, 17, 39, 34]) validates this assertion—the vast majority of real attacks (and all the attacks we saw that execute malicious code) involve inserting unanticipated `%n`-specifiers into format strings. Thus we choose to focus on `%n`-based attacks in this paper. However, our technique could easily be extended to handle other types of format-string attacks.

[1] In general, C's variable-argument functions cannot check this.

1.2 White-Listing

We propose a simple, flexible, and direct way to control the memory modified by a function (such as `printf`): An explicit, dynamic white-list of address ranges can control writes that may be unsafe, such as those exploited by format-string attacks. We can add and remove address ranges from the white-list at run-time, and we can require that the writes we wish to guard first check that an address is in the white-list before writing to it. White-lists can be viewed as a direct representation of a simple but easily understood security policy: Certain code should modify only certain locations. Of course, the white-list itself must not succumb to malicious modification, but such an event can occur only if an application is already compromised.

To see the flexibility of white-listing, notice that it can easily encode many static approaches:

- Default: A white-list containing the range $[0, 2^n - 1]$ (for an n-bit address space) allows any write, effectively turning off protection. This might be necessary, for example, to ensure that legacy code executes unchanged.
- Read-only: An empty white-list ensures that the vulnerable write it protects never successfully executes. For format strings, the empty white-list is equivalent to banning the `%n` format character.
- Sandboxing: A white-list containing just the range $[2^j, 2^{(j+1)} - 1]$ restricts writes to an aligned 2^j range, mirroring some approaches to software fault isolation [37].

Furthermore, because our white-lists are dynamic, we can express more interesting dynamic policies (such as the one we describe in this paper), and we can change the policy at run-time if desired.

In short, we believe dynamic white-lists are powerful tools for increasing the security of C code by restricting memory writes. In particular, in this paper we use the approach to prevent format-string attacks. We show that it is effective, easy-to-use, and efficient.

1.3 Contributions and Outline

We have implemented a white-list based approach to preventing format-string attacks, and determined that the performance overhead is reasonable. Moreover, our approach uses a simple static code analysis to detect user-defined wrapper functions that call `vprintf` and similar functions. This allows us to rule out common[2] attacks that previous work could not [6]. An explicit white-list lets us trade-off precision and security when the flow analysis is too conservative. Our approach does not rely on the C preprocessor

[2] A brief manual search found several known vulnerabilities involving wrapper functions that call `vfprintf`, including: isc dhcpd 3.0[30], rwhoisd 1.5[28], zkfingerd 0.9.1[25], unreal ircd 3.1.1[19], nn news reader[39], wu-ftpd[35], and tcpflow 0.2.0[34].

(we perform a source-to-source transformation on preprocessed code) and does not require recompiling the standard C library.

Section 2 shows how white-lists can prevent format-string attacks. In Section 3, we describe the automatic maintenance of the white-list. Section 4.1 presents vulnerability prevention results, and in Section 4.2 we examine the runtime overhead of white-listing. Section 5 contrasts our work with related projects. Finally, Section 6 concludes and considers other interfaces that may benefit from white-listing.

2. WHITE-LIST BASED PREVENTION OF FORMAT-STRING ATTACKS

Using a white-list to prevent format-string attacks is straightforward:

- We need a run-time white-list containing the address ranges that printing functions are allowed to write.
- The printing functions must consult the white-list before executing the code for a `%n` qualifier.
- Callers to printing functions must register (add to the white-list) any locations that might legally be written.
- For performance and security, callers should unregister locations after the printing functions are called.

We represent the white-list with a (resizable) array of address ranges. At any point during program execution, we expect the white-list will need to contain at most a few addresses (because of the fourth point above). We store the white-list in a global variable; in a multithreaded setting it would be slightly more secure to use a separate white-list for each thread.

To check the white-list, before executing a `%n` qualifier the printing function must first verify that the location it is about to write is in a registered address range. We expect the addresses registered at the most recent call site to be used first (we shall see why this is the case in Section 3). Thus, the function checks the white-list array as a stack, starting with the most recently registered range. We can either modify (or reimplement) the printing functions, or we can wrap them with a function that checks the white-list. The former has the advantage of performance (parsing the format string only once), but the disadvantage that we intrusively change or circumvent the standard library. If a white-list check fails, we choose to abort the program, but other choices are possible (such as silently skipping the write or sending a signal).

We provide a very simple API for users to adjust the white-list; richer interfaces are certainly possible but we have not needed them.

- `__register(x,y)` adds the range $[x, y)$ to the white-list.
- `__register_word(x)` adds the range $[x, x]$ to the white-list. We can use this function to register addresses pointed to by arguments that are meant to be used as `%n` targets.
- `__unregister()` removes the most recently added-but-not-yet-removed range from the white-list (i.e., it pops the stack). This is more efficient than searching the white-list for a specific address, and has been sufficient for our purposes. In particular, it is exactly what we need for the automated approach described in Section 3.

For programs that never use `__register`, we can implement a simpler white-list by merely disabling the `%n` modifier. Calling `__unregister()` on an empty white-list has no effect.

For security, clients should register as few ranges as possible, and unregister them as soon as possible. Ideally, that would mean registering the correct arguments just before calling a printing function and unregistering them immediately afterward. However, wrapper functions that pass `va_list`s to functions like `vfprintf` (e.g., `wrapper1()` in Figure 1) cannot do the registration: They do not know the number or types of the arguments in the list. Therefore, the function deeper in the call stack that *does* know this information must do the registration (i.e., we must register at the call site of the function whose arguments are pointed to by the `va_list`). As the next section shows, we can automate this process with a simple static analysis.

3. AUTOMATIC WHITE-LIST MAINTENANCE

We have implemented a fully automatic source-to-source transformation that inserts calls to `__register_word` (and `__unregister`) for any argument of type `int*` or `unsigned*` that might be passed to a printing function (either directly or via a `va_list`).[3] Section 3.1 describes the basic approach. Section 3.2 discusses trickier issues (function pointers and non-local `va_list` values). Section 3.3 considers separate compilation (when not all source code is available). Section 3.4 describes some important automatic optimizations for avoiding unnecessary API calls. Finally, Section 3.5 considers manual optimizations.

3.1 Basic Approach

We built our transformation using the freely available CIL [20, 4] system for manipulating C programs. CIL takes a program, performs a number of meaning-preserving simplifying transformations, and produces an Abstract Syntax Tree (AST). The CIL system is easily extensible, and provides code to simplify the process of creating static analyses and source transformations. The CIL system also has the capability to merge separate source files (with the `--merge` command line flag), which greatly simplifies the construction of whole-program analyses. For our purposes, whole-program analysis is not strictly necessary, but leads to more precise results (see Section 3.3). Extending CIL also makes it easy for programmers to use our approach: they can simply use `cilly --merge --doautoWhitelist` as their "C compiler", and the transformation and subsequent C compilation will happen seamlessly.

[3] We chose the more secure policy here of only registering integer pointer arguments, because we felt that it was unlikely that programmers would intend to use other types with a `%n`. Our mechanism will also support more permissive policies that register other types of arguments. There is a trade-off here, however, because more registration means a larger white-list, which in turn means less security.

The actual transformation we perform involves function calls: For each call `e0(e1,e2,...,en)` and argument `ei` (where $i \geq 1$), we determine if:

1. `ei` is passed to the callee as a variable argument (which we can tell from the type of `e0`)
2. `ei` is a pointer to an integer (which we can tell from the type of `ei`)
3. `e0` is a printing routine or its variable arguments may be passed to a printing routine.

If all three facts hold, we surround the function call with `__register_word(ei)` and `__unregister()`. Of course, we may register and unregister multiple arguments. Figure 1a contains an example program, with `printf` calls and with wrapper functions calling `vprintf`. We also include a `printf` with a constant format string. This call is not rewritten nor are its arguments registered, as explained in Section 3.4. Figure 1b shows the result of transforming the example with our tool.

Because we duplicate the registered expressions, we must be sure that they have no side-effects. For example, if we need to register argument `p++`, we must ensure that our transformation does not increment the pointer `p` twice. Fortunately, we can exploit a transformation that already happens automatically in CIL. This transformation replaces all function arguments with effect-free expressions (such as variables). For example, `foo(p++)` will get transformed to:

```
tmp = p++;
foo(tmp);
```

Our transformation will then register the temporary variable, rather than `p++`, thus avoiding the duplication of side effects.

The only necessary question that is not directly answered by CIL (or by any system that type-checks C code) is whether fact (3) holds. The arguments passed to a printing function may come from two different sources. They may be supplied directly at the call site (as in the case of `printf`, `syslog`, `fprintf`, etc.), or they may be part of another function's variable-argument list, and passed via a `va_list` (as in the case of `vprintf`, `vsyslog`, `vfprintf`, etc.). If the arguments are supplied directly, we can simply register (and unregister) them at the call site. However, if the arguments are passed from elsewhere, we must determine which functions they could have come from. We can then register arguments at the call sites of those functions. We identify those functions with a conventional, whole-program dataflow analysis. As Section 4.2 shows, this analysis is tractable even for large programs.

In particular, our analysis determines the contents of two sets of functions. The first set, $\mathcal{S}_1$, contains exactly those functions whose integer-pointer arguments must be registered. Specifically, these are the variable-argument printing functions (`printf`, `sprintf`, etc.), and the functions that create a `va_list` that may be passed to a `va_list` printing function (`vprintf`, `vsprintf`, etc.). The second set, $\mathcal{S}_2$, contains those functions that take a `va_list` as an argument, and that may eventually pass that `va_list` to a `vprintf`-style function. They may either pass the `va_list` directly, or they may pass it through other functions that (transitively) pass it to `vprintf`. Note $\mathcal{S}_2$ also includes the `vprintf`-style functions themselves.

Our analysis proceeds as follows. We initialize $\mathcal{S}_1$ with the `printf`-style functions, and $\mathcal{S}_2$ with the `vprintf`-style functions. We then iterate over every function in the program. For each function `f`:

1. If `f` is a variable-argument function that passes a `va_list` to a function in $\mathcal{S}_2$, then we must add `f` to $\mathcal{S}_1$. (The function in $\mathcal{S}_2$ may pass `f`'s arguments to `vprintf`, so we must register them.)
2. If `f` has a `va_list` argument and calls a function in $\mathcal{S}_2$, then we must add `f` to $\mathcal{S}_2$. (The function in $\mathcal{S}_2$ may pass `f`'s `va_list` to a `vprintf`, so `f` needs to be in $\mathcal{S}_2$ as well.)

The above cases handle `va_lists` that are created in `f`, or passed into `f`. The other possibility—that the `va_list` is drawn from some data structure—is discussed in Section 3.2. We repeatedly iterate over all the functions until no new functions are added to either set. At this point, $\mathcal{S}_2$ will contain all the functions that can (transitively) pass a `va_list` argument to a `vprintf`-style function. The set $\mathcal{S}_1$ will contain all the variable-argument functions that call any of the functions in $\mathcal{S}_2$, plus the original variable-argument printing functions. Thus $\mathcal{S}_1$ will contain the functions whose arguments must be registered. Note that there is nothing unusual here: It is a "textbook example" of a dataflow analysis. We also expect it to be tractable for even the largest programs because only functions with a variable number of arguments or an argument of type `va_list` are relevant: We can precompute that the vast majority of functions are irrelevant and need not be considered when the analysis iterates.

Having precomputed the two sets as just described, it is trivial to determine if part (3) holds, provided `e0` is the name of a function. Specifically, part (3) holds if `e0` is in $\mathcal{S}_1$.

In practice, this basic approach has sufficed for every application we have investigated. Nonetheless, it is not quite sufficient for arbitrary C programs, or for programs where some of the source code is unavailable. We now consider these complications.

3.2 Function Pointers and Data Structures

In the previous section, we assumed that function expressions (i.e., `e0`) were function names, and that every `va_list` was either a local variable or a function argument. In practice these assumptions hold for printing functions, but in theory they might not.

If we cannot statically determine the function pointed to by a variable-argument function pointer, we can instead "guess" whether or not its arguments should be registered. Guessing yes means we will not risk aborting a program that is using `%n` correctly. On the other hand, guessing no is more secure because the white-list stays smaller. Our current system errs on the side of more security, and thus always guesses no. However, this is a question of *policy*—our *mechanism* can support either choice.

We can also do arbitrary things with a `va_list`, such as storing it in a data structure or a global variable. In this case, we may not be able to determine at compile-time whether the program might subsequently pass the `va_list` to a printing routine; so again we "guess". As before, we err on the side of more security and guess no. However, our mechanism could easily support a policy of guessing yes. We would simply treat any function that assigns a `va_list` to a

```
void wrapper1(char *fmt, va_list args){
   vprintf(fmt, args);
}

void wrapper2(char *fmt, ...){
   va_list args;
   va_start(args, fmt);
   wrapper1(fmt, args);
   va_end(args);
}

int main(int argc, char **argv){
   char str[100] = "Hello%n world!";
   int x;
   int *y = &x;
   printf(str, y);
   wrapper2(str, y);
   printf("Hello world!");
}
```

(a)

```
void wrapper1(char *fmt, va_list args){
   __vprintf_Checked(fmt, args);
}

void wrapper2(char *fmt, ...){
   va_list args;
   va_start(args, fmt);
   wrapper1(fmt, args);
   va_end(args);
}

int main(int argc, char **argv){
   char str[100] = "Hello%n world!";
   int x;
   int *y = &x;
   __register_word(y);
   __printf_Checked(str, y);
   __unregister();
   __register_word(y);
   wrapper2(str, y);
   __unregister();
   printf("Hello world!");
}
```

(b)

Figure 1: Example code (a) before and (b) after automatic white-listing.

"strange place" (such as a `struct` field or global variable) as though that function might call `vprintf` with the `va_list`. Doing so will cause our analysis to trace the `va_list` back to its source.

3.3 Separate Compilation

Our iterative program analysis assumes that all the source code is available (except for the standard library). CIL makes it easy to provide all the source code, as long as it is available to the programmer. If it is not, then we cannot know whether a function taking variable arguments (or a `va_list`) might use them for printing. Programmers can write stub functions to provide the answer, or the tool can "guess". Once again, we choose to err on the side of more security and guess no. However, as before, this is a matter of the policy that we chose for our prototype. The mechanism of white-listing can easily support either policy choice. Similarly, annotations such as gcc's `printf` attribute can guide the guesses.

3.4 Optimizations

In many cases, we can avoid the run-time overhead associated with registration and white-list checking. In particular, we can circumvent registration and white-list checking whenever the format string is immutable. Format string attacks involve inserting unexpected format specifiers into user-supplied format strings. Thus static format strings are not vulnerable.

Alternatively, if we also wanted to provide protection against incorrect uses of `%n`-specifiers by programmers, we could instead treat constant format strings as follows: For each `%n`-specifier (typically none) in the format string, check statically that the corresponding actual argument is an integer pointer. Then, as above, circumvent registration and white-list checking.

The constant-string optimization requires two versions of each `printf`-style function (one that checks the white-list and one that does not). However, as we will see in section 4.2, the performance benefit more than makes up for this small amount of code duplication.

Another possible optimization, which we have not yet implemented, would be to transform calls where no arguments follow the format string into calls with a constant format string. In particular, we can replace calls of the form `printf(buf)` with `printf("%s", buf)`, as long as no arguments appear after `buf`. If `buf` contains no format specifiers, these two calls are identical. On the other hand, if `buf` does contain format specifiers, the behavior of `printf(buf)` is undefined (because there are no corresponding arguments). Thus a C compiler can do whatever it chooses—including the proposed replacement.[4] The new form has a constant format string, and thus we can apply the previous optimization.

3.5 Manual registration

Explicit white-list maintenance allows the programmer to control the cost of registering address ranges. For example, consider this code fragment, where we assume that each element of `arr` is a string of the form `"... %d ... %n"`:

```
int total = 0, x = 0, i = 0;
for(; i < arr_len; ++i) {
  __register_word(&x);
  printf(arr[i], i, &x);
  __unregister();
  total += x;
}
```

[4]It is debatable whether using program rewriting to mask a potential error is worthwhile. In the interest of security, we believe it is.

```
int main(int argc, char **argv) {
   int i;
   char buf[50];
   for (i=0; i < 10000000; i++) {
      sprintf(buf, "butter");
   }
}
```

Figure 2: The first performance microbenchmark. In this test, the format string contains no format specifiers.

```
int main(int argc, char **argv) {
   int i,j,k;
   char buf[50];
   for (i=0; i < 10000000; i++) {
      sprintf(buf, "butter%d%d", j, k);
   }
}
```

Figure 3: The second performance microbenchmark. In this test, the format string contains two format specifiers.

Our automated process would produce the above code, but the following is more efficient:

```
int total = 0, x = 0, i = 0;
__register_word(&x);
for(; i < arr_len; ++i) {
  printf(arr[i], i, &x);
  total += x;
}
__unregister();
```

Notice that we have hoisted the registration and unregistration out of the loop, so the program executes them only once. It is conceivable that an optimizing compiler could also detect this optimization opportunity. However, there will always be cases where the programmer knows more than the compiler, and thus can do a better job placing registrations.

Although formatted output is rarely performance-critical, this feature of white-lists may be more important for other applications. Thus, our tool allows programmers to turn off automatic registration and instead insert their own calls. If a programmer forgets to register an argument the program may abort, but it will not compromise security.

4. RESULTS

In this section we present performance results for our white-listing format-string tool. Section 4.1 discusses our effectiveness at preventing format-string vulnerabilities, and Section 4.2 presents our run-time performance overhead.

In both sections, we compare our results with FormatGuard [6]. FormatGuard is similar to our approach in that it combines compile-time source transformations with run-time checks (see Section 5.1 for more details). It has proven effective at preventing many format-string vulnerabilities.

4.1 Vulnerability Prevention

We tested our approach on four programs with known format string vulnerabilities:

```
int main(int argc, char **argv) {
   int i,j,k;
   char buf[50];
   for (i=0; i < 10000000; i++) {
      sprintf(buf, "butter%n%n", &j, &k);
   }
}
```

Figure 4: The third performance microbenchmark. In this test, the format string contains two '%n' format specifiers.

- The `tcpflow` program is frequently run as root, and can be attacked by inserting format specifiers into specific command-line arguments [34]. Thus an ordinary user can obtain a root shell via this exploit.
- The `splitvt` program is also typically run as root, and can also be attacked by inserting format specifiers into one of its command line arguments [17]. Thus `splitvt` is another potential source of unauthorized root shells.
- The `rwhoisd 1.5` [28] server[5] is vulnerable to format specifiers embedded in strings that follow the `-soa` directive. Attackers can use this vulnerability to gain a remote shell on the machine running `rwhoisd`.
- The `pfinger` client is vulnerable to format specifiers in remote `.plan` files [24].

Our white-listing approach fixes all these vulnerabilities. The FormatGuard approach, on the other hand, fixes only the `splitvt` and `pfinger` vulnerabilities.

The key difference between the power of our approach and the power of the FormatGuard approach is that we are able to prevent attacks on `vprintf` format-strings, such as the attacks on `tcpflow` and `rwhoisd` mentioned above. We feel this additional expressiveness is vital, because a number of well-known format-string attacks specifically involve `vprintf`-style functions. In addition to the two already mentioned, these include the famous `wu-ftpd` vulnerability [35] (the first well-known format-string vulnerability), and the vulnerabilities in `isc dhcpd 3.0` [30], `zkfingerd 0.9.1` [25], `unreal ircd 3.1.1` [19], and the `nn` news reader [39].[6] Additionally, a system where `vprintf` is insecure discourages the use of wrapper functions for logging and I/O—which is generally considered good software engineering practice.

4.2 Performance Overhead

To determine our overhead per `printf` call, we ran a series of simple microbenchmarks consisting of a single loop containing a single `sprintf` call. We also downloaded a copy of FormatGuard, and compared our overhead with its overhead. The tests were run on a 2.26 GHz Pentium 4,

[5]We had to modify the `rwhoisd` code slightly, because it would not compile with the version of `gcc` installed on our machines (version 3.3.3). We emphasize that these modifications were necessary because of `gcc`, and not because of our tool.

[6]We were unable to test our approach on all of these vulnerabilities, however, because updated versions have been released (and the old source code is no longer available).

with 500 MB of RAM, and compiled with `gcc` version 3.3.3 using no compile flags.[7] The tests were all run with the constant string optimization disabled (otherwise the white-listing overhead would have been 0%, because our micro-benchmarks use only constant strings). We also chose to wrap calls to `printf`-style functions with whitelist-checking functions,[8] rather than to reimplement the `printf` functions. Reimplementing `printf` would have likely led to better results, but it would also lead to an unfair comparison with FormatGuard—because they also chose to wrap functions rather than reimplement them.

Our performance varied with the number and types of the specifiers in the format string. This was expected: The overhead of white-list checking is proportional to the number of specifiers. With no specifiers (Figure 2), white-listing added an overhead of 10.2% and FormatGuard added an overhead of 7.5%. With two non-'`%n`' specifiers (Figure 3), our approach added an overhead of 28.6%, and FormatGuard added 20.9%. With two '`%n`' specifiers (Figure 4), our overhead was 60.0%, and FormatGuard's was 38.1%.

We also tested `vsprintf` by moving the printing loop inside a wrapper function. We observed an overhead of 26.4% with no specifiers, 39.8% with two non-'`%n`' specifiers, and 74.7% with two '`%n`' specifiers. FormatGuard does not protect against `vsprintf` vulnerabilities, and thus does not transform these benchmarks. Note that the `vsprintf` overhead percentages are exaggerated relative to the `sprintf` overheads because `vsprintf` executes faster than `sprintf`. The results for all our microbenchmarks are summarized in Figure 5.

These overheads may seem high, but we stress that these are microbenchmarks and not realistic programs. In addition, we had to turn off our constant-string optimization. As we show below, this optimization can often significantly reduce the overhead.

We also searched for a real, `printf`-intensive application to test our performance. We had some difficulty finding an application where `printf` was performance critical, because most I/O intensive programs implement their own I/O procedures. We eventually settled on `man2html`, the same program used by the FormatGuard authors to test their performance. As the name suggests, the `man2html` program converts man pages to HTML web pages. We used the same machine and compiler that we used for the microbenchmarks. With the constant format string optimization turned off, our approach added an overhead of 14.1%. With the optimization enabled, our approach added only 0.7%. The FormatGuard approach added an overhead of 9.0%. As we see, our optimization allowed us to execute this `printf`-heavy application with insignificant overhead. Our performance with the optimization enabled was also noticeably better than FormatGuard, even though they catch a smaller class of vulnerabilities.

There are two likely reasons that would explain why we experienced a higher overhead than FormatGuard on the microbenchmarks, and in the `man2html` test without optimizations. Both tools use the `parse_printf_format` function to parse the format strings, but our approach must do more with the result of the parse. FormatGuard only needs a count of the number of format specifiers (returned by `parse_printf_format`), whereas we need to actually look at each specifier to determine if it is a `%n`. In addition, we must pay the extra cost of registering (and unregistering) pointer arguments.

However, as we saw when we turned on the constant-string optimization, our overhead on "real-world" applications is insignificant. This is further borne out by performance tests we ran on the applications we mentioned in Section 4.1 (once again, with the same machine and compiler). We ran `rwhoisd` in local mode (to avoid network delays), and observed an average overhead of 1.3% to start-up and respond to a query (1.6% without optimization). We ran `tcpflow` over a 273MB `tcpdump` output file and observed an overhead of 0.3% (0.9% without optimization). We also attempted to test the `pfinger` client, but found that the variability due to network delays drowned out any difference between the white-listed and normal versions. We did not test the overhead of `splitvt`, as we could not think of a sensible test.[9] In general, these results confirm our belief that many vulnerable C applications do not benefit from the performance gained by using insecure library facilities. Figure 5 summarizes these results.

We also measured the compile-time overhead of our approach (with the same machine and compiler as above). These results are summarized in Figure 6. We see that using CIL added a significant overhead to each compile, but that our analysis and transformation added only a small additional overhead—between 0 and 4.7%. We implemented our prototype using CIL because it was much simpler to extend, but these results suggest that an implementation could gain significant compile time savings by instead integrating directly into the compiler. The overhead when we use CIL is primarily caused by parsing and typechecking the program twice (once by CIL, and then again by `gcc` when it compiles the transformed program produced by CIL). If we instead integrated our approach directly into the compiler, we would need to parse and typecheck the program only once.

5. RELATED WORK

Our approach to preventing format-string attacks nicely complements other approaches:

- It prevents more attacks than FormatGuard (Section 5.1).
- It rejects fewer safe programs than approaches preventing format arguments from "tainted" sources (Section 5.2).
- It is more efficient than approaches that check all writes in an application (Section 5.3).
- It is more efficient and less intrusive than approaches that change the variable-argument calling convention (Section 5.4).

[7] We tried compiling our microbenchmarks with optimizations enabled, but found that this actually slowed down the code, with or without white-listing. This occurred regardless of whether we used `-O1`, `2`, or `3`.

[8] Our wrapped functions ensure correct parsing by using the glibc function `parse_printf_format`—the same function that the actual printing functions use to parse their format strings.

[9] The `splitvt` application simply splits a terminal into two terminals. There is no noticeable delay when interacting with the terminals, either with or without white-listing.

Benchmark	White-listing	Optimized White-listing	FormatGuard
`sprintf` microbenchmark, no specifiers	10.2%	0% (see below)	7.5%
`sprintf` microbenchmark, 2 %d specifiers	28.6%	0% (see below)	20.9%
`sprintf` microbenchmark, 2 %n specifiers	60.0%	0% (see below)	38.1%
`vsprintf` microbenchmark, no specifiers	26.4%	0% (see below)	no protection
`vsprintf` microbenchmark, 2 %d specifiers	39.8%	0% (see below)	no protection
`vsprintf` microbenchmark, 2 %n specifiers	74.7%	0% (see below)	no protection
`man2html`	14.1%	0.7%	9.0%
`rwhoisd`	1.6%	1.3%	no protection
`tcpflow`	0.9%	0.3%	no protection

Figure 5: Performance results comparing the overhead of white-listing format strings (both with and without the constant string optimization) with the overhead of FormatGuard. The microbenchmarks have no overhead with optimized white-listing, because their format strings are constant.

Benchmark	Source Lines	gcc 3.3.3	gcc + CIL	gcc + CIL + Whitelisting	White-listing Overhead
`splitvt`	5288 lines	1.85 sec.	2.91 sec.	2.94 sec.	1.0%
`pfinger`	331 lines	0.15 sec.	0.36 sec.	0.36 sec.	0%
`man2html`	3630 lines	0.60 sec.	1.14 sec.	1.15 sec.	0.9%
`rwhoisd`	29702 lines	7.95 sec.	19.18 sec.	20.09 sec.	4.7%
`tcpflow`	1695 lines	0.67 sec.	1.14 sec.	1.16 sec.	1.8%

Figure 6: The time required to compile the macrobenchmarks with gcc 3.3.3, with gcc and CIL, and with gcc and CIL extended with our white-listing analysis and transformations. The final column indicates the overhead due to white-listing (i.e., it compares the fourth and fifth columns).

5.1 FormatGuard

FormatGuard [6] cleverly uses features of the Gnu C Preprocessor to count arguments to `printf`-style functions. It then calls wrapper functions that reject calls with too many format specifiers. This approach relies on the fact that the number of arguments is known at compile-time. As discussed in the previous section, the run-time overhead is comparable to our approach. FormatGuard is simpler in that it requires no compile-time flow analysis, but it has corresponding limitations.

Most importantly, it does not detect attacks on `vprintf`-style functions. Not only are such attacks common, but a system in which `printf` is more secure than `vprintf` discourages the good practice of interceding output with application-specific functions. On the other hand, FormatGuard could probably be extended to allow programmers to explicitly declare their printing functions.

The FormatGuard approach will also miss an attack that replaces another format specifier with `%n`. We are unaware of any such attacks; they seem quite difficult to construct.

FormatGuard also lacks an analogue of our constant-string optimization. It may be possible to add such an optimization to FormatGuard by using `gcc`'s builtin function `__builtin_constant_p` (or something similar). In our system, however, the optimization was trivial to implement because of the type-checked abstract-syntax tree provided by CIL.

5.2 Tainted-String Detection

A compile-time or run-time analysis can detect whether a format string passed to a printing function could possibly have come from an untrusted source. These format strings can then be rejected. Different systems use different analyses and different definitions of untrusted sources.

Shankar et. al. [31] and Guyer et. al. [12] use compile-time flow analyses to identify and track strings that may have come from I/O or that may have been modified by the user. If this potentially tainted data is used as the format string of a `printf` or `syslog` call, an error is declared. This approach is more conservative than necessary because static analysis is inherently limited and much supposedly "tainted" data is actually perfectly safe. In addition, purely static-analysis based techniques like these require changing the code to fix any potential vulnerabilities that are found—which can be difficult when dealing with large applications that you do not understand well (e.g., if you are trying to protect open-source code that you are compiling and installing on your machine). With our approach, however, the protection is automatic—the user does not have to change any code.

Other compile-time approaches are less complete. For example, Alan DeKok's PScan [9] finds `printf` call sites where the format string is both non-static and the final parameter. The `gcc` compiler [11] flag `-Wformat=2` causes the compiler to issue a warning whenever a non-static format string is found. These approaches give warnings about safe code (false positives) and can miss format-string vulnerabilities (false negatives).

At run-time, we can detect suspicious format strings or writes that corrupt function return addresses. The `libformat` library, by Tim Robbins [27], takes the former approach. It rejects any `printf` that uses a format string

that is in writable memory and that contains a `%n`-specifier. This is essentially equivalent to using an empty white-list in conjunction with our constant string optimization. This approach may abort safe and correct programs, including the example in Figure 1. The `libsafe` library of Tsai and Singh [36] takes the latter approach, verifying that the write caused by a `%n` is not to a function return address. It is equivalent to white-listing all of memory except the locations of the return addresses. This approach might miss less direct format-string attacks. The implementation also requires frame pointers, i.e., it is incompatible with gcc's `-fomit-frame-pointer` flag.

5.3 Restricting Writes

One can view our explicit white-list as a software approach to restricting a memory write based on the address being written. Related work has taken a similar approach for an entire application, rather than a specific vulnerability such as format-string attacks.

Software fault isolation (SFI) [37] is one such approach. The legal address range is chosen ahead of time and the binary code rewritten to enforce the restriction efficiently. Unfortunately, a compile-time white-list is not flexible enough to prevent format-string attacks.

Systems like Safe-C [1, 16] are more flexible than SFI, checking at run-time that writes to memory do not violate array bounds, follow dangling pointers, etc. This approach can slow down applications by as much as an order of magnitude, making it inappropriate in many settings. On the other hand, our white-list approach is relevant to only printing functions, making the performance overhead more than reasonable.

5.4 Safe `printf`

CCured [21, 5] (a type-safe implementation of C) and Cyclone [14, 8] (a type-safe dialect of C) take similar approaches to making the printing functions safe. Roughly, the caller provides the number and types of the variable arguments, and the callee compares them against the format string at run-time. The compiler does this implicitly at call sites so there is no burden for the programmer using the printing functions.

In Cyclone, no attempt is made to preserve the native calling convention for variable-argument functions. Instead, a stack-allocated array holds the variable arguments and their type tags. In CCured, the type information can be passed via a global variable. In both cases, there is extra data (the type tags) and extra parameter passing, even when the format string does not contain `%n`. Despite using a flow analysis, our approach is simpler and more efficient than making C type-safe.

6. CONCLUSIONS

We have presented a solution to the problem of format-string attacks, by providing an automated approach to maintaining an explicit white-list. We have found that a white-list directly encodes the relevant security policy, namely that printing functions should modify only certain caller-specified memory locations. Furthermore, a dynamic white-list provides flexibility: We can change the policy at run-time and we can directly encode common policies ranging from "no checking" to sandboxing to "no writes." With little performance overhead, our approach has fewer false positives and fewer false negatives than previous work. In particular, we catch attacks using `vprintf` and we do not forbid the `%n` format specifier in non-static format strings.

An efficient, automatic, whole-program static analysis and transformation performs white-list maintenance without burdening programmers. The analysis is simple and efficient using the right tool (such as CIL, which provides a type-checked abstract-syntax tree), but would be impossible with macros or simple scripts. Moreover, we use the analysis just to insert the correct run-time checks, so the imprecision of static analysis is not a limitation.

We believe white-lists are a useful tool for implementing software security policies and consider this work a compelling example. We look forward to considering white-lists for reducing other security vulnerabilities. Specific examples include preventing race conditions for file I/O [3], limiting references to kernel data in user buffers, and restricting access to communication ports.

7. ACKNOWLEDGMENTS

We would like to thank Steve Gribble, Michael Hicks, Jesse Rothstein, and the anonymous reviewers for helpful comments on earlier drafts.

8. REFERENCES

[1] Todd Austin, Scott Breach, and Gurindar Sohi. Efficient detection of all pointer and array access errors. In *ACM Conference on Programming Language Design and Implementation*, pages 290–301, Orlando, FL, June 1994.

[2] William Bush, Jonathan Pincus, and David Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30(7):775–802, June 2000.

[3] Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of C code. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, 2004.

[4] *CIL - Infrastructure for C Program Analysis and Transformation, version 1.3.2.* Available at *http://manju.cs.berkeley.edu/cil/*.

[5] Jeremy Condit, Matthew Harren, Scott McPeak, George Necula, and Westley Weimer. CCured in the real world. In *ACM Conference on Programming Language Design and Implementation*, pages 232–244, June 2003.

[6] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. FormatGuard: Automatic protection from `printf` format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Washington, D.C., Aug. 2001.

[7] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *7th USENIX Security Symposium*, pages 63–78, San Antonio, TX, January 1998.

[8] *Cyclone, version 0.8.* Available at *http://www.research.att.com/projects/cyclone*.

[9] Alan DeKok. Pscan: A limited problem scanner for C source files, July 2000. Available at *www.striker.ottawa.on.ca/~aland/pscan/*.

[10] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *4th USENIX Symposium on Operating System Design and Implementation*, pages 1–16, San Diego, CA, October 2000.

[11] Free Software Foundation. The GNU compiler collection. Available at *http://gnu.gcc.org/*.

[12] S. Z. Guyer, E. D. Berger, and C. Lin. Detecting errors with configurable whole-program dataflow analysis. Technical Report UTCS TR-02-04, UT-Austin, 2002.

[13] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Winter USENIX Conference*, pages 125–138, San Francisco, CA, January 1992.

[14] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, Monterey, CA, June 2002.

[15] Stephen Johnson. Lint, a C program checker. Computer Science Technical Report 65, Bell Laboratories, December 1977.

[16] Richard Jones and Paul Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *AADEBUG'97. Third International Workshop on Automatic Debugging*, volume 2(9) of *Linköping Electronic Articles in Computer and Information Science*, 1997.

[17] Michel Kaempf. Multiple vulnerabilities in splitvt, January 2001. At *www.securityfocus.com/archive/1/156251*.

[18] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, pages 191–206, August 2002.

[19] Gabriel A. Maggiotti. Unreal ircd format string vuln, February 2002. At *www.securityfocus.com/archive/82/258190*.

[20] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the Conference on Compiler Construction*, pages 213–228, 2002.

[21] George Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *29th ACM Symposium on Principles of Programming Languages*, pages 128–139, January 2002.

[22] T. Newsham. Format string attacks. White Paper, Sept. 2000. At *www.securityfocus.com/guest/ 3342*.

[23] Bruce Perens. Electric fence. At *www.gnu.org/directory/All_Packages_in_Directory/Electric-Fence.html*.

[24] NGSSoftware Insight Security Research. Pfinger 0.7.8 format string vulnerability, December 2002. *http://www.securityfocus.com/archive/1/303555*.

[25] NGSSoftware Insight Security Research. zkfingerd 0.9.1 format string vulnerability, December 2002. *http://www.securityfocus.com/archive/1/303557*.

[26] Michael F. Ringenburg and Dan Grossman. *www.cs.washington.edu/homes/miker/formatstring/*.

[27] Tim Robbins. libformat, November 2001. At *www.wiretapped.net/~fyre/software/libformat.html*.

[28] Rwhoisd remote format string vulnerability, October 2001. At *www.securityfocus.com/archive/1/ 222756*.

[29] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[30] VOID.AT Security. isc dhcpd 3.0 format string exploit, January 2003. At *www.securityfocus.com/archive/1/306327*.

[31] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *10th USENIX Security Symposium*, pages 201–220, 2001.

[32] Christopher Small and Margo Seltzer. MiSFIT: constructing safe extensible systems. *IEEE Concurrency*, 6(3):33–41, July–September 1998.

[33] Splint manual, version 3.0.6, 2002. *http://www.splint.org/manual/*.

[34] @stake, Inc. tcpflow 0.2.0 format string vulnerability, August 2003. At *www.securityfocus.com/advi-sories/5686*.

[35] tf8@zolo.freelsd.net. Wu-ftpd remote format string stack overwrite vulnerability, June 2000. At *www.securityfocus.com/bid/1387*.

[36] T. Tsai and N. Singh. Libsafe: Protecting critical elements of stacks. Technical Report ALR-2001-019, Avaya Labs, Aug. 2001.

[37] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 7(5):203–216, December 1993.

[38] J. Wilander and M. Kamkar. A comparison of publicly available tools for static intrusion prevention. In *Proceedings of the 7th Nordic Workshop on Secure IT Systems*, pages 68–84, Nov. 2002.

[39] zillion. nn format string exploit, July 2002. *http://www.securityfocus.com/archive/82/280687*.