Preventing Variadic Function Attacks Through Argument Width Counting

Brennan Ward*, Kevin Dennis*, Gabriel Laverghetta, Parisa Momeni, and Jay Ligatti

Abstract Variadic-function attacks and format-string attacks continue to threaten modern C/C++ software applications. When successfully executed, attackers are able to read, write, or execute arbitrary program memory. To address the problem of variadic-function attacks, this paper proposes Argument Width Counting (AWC), a new memory access-control policy that, when enforced, mitigates all observed variadic-function attacks, including format-string attacks, by tracking the initial size of variadic arguments on the stack and limiting requests to this number of bytes. A prototype for AWC has been implemented and evaluated on LLVM's Clang C/C++ compiler and the accompanying libc++ standard library. The implementation modifies the compiler's function-generation code to store variadic argument widths and to validate these values later when variadic arguments are accessed. The prototype's performance overhead was tested and compared to existing solutions. The prototype incurs low overhead and outperforms the existing solutions. Microbenchmarking AWC returns around 22% overhead for 0 to 16 arguments. The overhead is less than 1% when benchmarked on real-world programs.

Key words: Variadic functions, format-string attacks, memory safety

---

Brennan Ward
University of South Florida, Tampa, FL, USA, e-mail: bward7864@gmail.com

Kevin Dennis
University of South Florida, Tampa, FL, USA, e-mail: kevindennis@usf.edu

Gabriel Laverghetta
University of South Florida, Tampa, FL, USA, e-mail: glaverghetta@usf.edu

Parisa Momeni
University of South Florida, Tampa, FL, USA, e-mail: parisamomeni@usf.edu

Jay Ligatti
University of South Florida, Tampa, FL, USA, e-mail: ligatti@usf.edu

* The first two authors contributed equally to this work.

1 Introduction

Variadic functions, or functions that take a variable number of arguments, are susceptible to attack due to the lack of bounds checking when accessing variadic arguments in the C and C++ programming languages.

Format-string attacks, a subclass of variadic-function attacks, endure as a serious cybersecurity threat and continue to be reported in the CVE [16] every year. These attacks pose significant risks, including privilege escalation, remote code execution, and unauthorized data disclosure, equating their severity to that of buffer overflows [1,3,7,9,16,18]. Given their persistent occurrence, and their high severity, the imperative for further mitigations is evident.

Numerous countermeasures and mitigation techniques have been proposed, ranging from simple compiler warnings to type checking for variadic functions. These solutions seek to mitigate attacks without significantly compromising performance and usability. Unfortunately, only compiler warnings have been widely adopted. While the exact reason such techniques have not been more widely adopted is unclear, it may be due to the performance cost: more precise solutions may be too costly to implement, and less precise solutions, while more performant, may not mitigate enough attacks to justify even the smaller cost.

This paper presents Argument Width Counting (AWC), a new memory access control policy that, when enforced, prevents all observed types of variadic-function attacks, including format-string attacks. Unlike existing mitigations, AWC restricts argument access to the intended region of memory by calculating the sizes of arguments and tracking the consumption of memory. Once a variadic function has requested all of the bytes that have been allocated, subsequent attempts to access variadic arguments will be treated as a policy violation and will result in program termination (or, depending on implementation, a raised exception for C++).

AWC can be efficiently enforced at runtime with no major compiler modifications; AWC requires no changes to a target program's source code. The necessary calculations to enforce AWC are not computationally expensive, and a prototype implementation of AWC for Clang, a C compiler built on the LLVM code generator and optimizer [15], was found to incur low overhead and outperform the existing lightweight solutions. The prototype implementation and benchmarking suite is available on GitHub [4,5].

This paper is organized as follows. Section 2 reviews the necessary background information and defenses against format-string and variadic-function attacks that are present in the literature. Section 3 introduces and describes Argument Width Counting. Section 4 describes a successful implementation of AWC for Clang. Section 5 presents the results of an empirical analysis run on the implementation. Section 6 discusses the results of the empirical evaluation and our attempt to add AWC to GCC. Section 7 concludes and explores areas of future work.

2 Background and Related Work

A vast number of C programs are compiled for the x86 architecture, and the behavior of the stack plays an important role in allowing variadic-function attacks to take place. When a function call begins, a portion of the stack space, known as a stack frame, is allocated on the stack. The stack frame holds all of the locally relevant data for the function such as the arguments and local variables. Other calling conventions may be adapted to this work with minor adjustments (e.g., accounting for optimizations like placing the first several arguments in registers). When a function call takes place, the caller prepares the stack by pushing, in order, the function arguments (in reverse order from right to left) and the return address. Once the stack setup has concluded, program execution is transferred to the callee, which will store the current value of `EBP` (a register holding the stack base pointer) and allocate space for any local variables. Figure 1 shows an example of a prepared stack frame using the cdecl calling convention [8]. Upon conclusion of the function, the stack frame is discarded and execution returns to the caller.

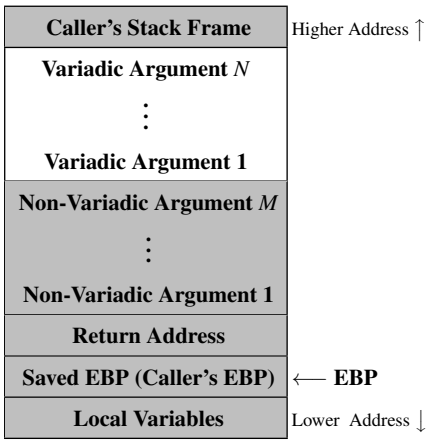

Fig. 1: An example stack frame, with data that may be accessed by `va_arg` in white

While the ISO C standard provides a standard interface for accessing variadic arguments, it does not provide a specific implementation or mechanism for the macros [14]. The underlying implementation is dependent on both the platform and compiler as each platform has slightly different calling conventions (although in practice the differences between compilers are likely negligible). The standard requires variadic arguments to be accessed through a struct called `va_list` and a set of macros. The `va_list` struct records data about the next variadic argument to be accessed. The struct internals do not need to be manipulated to enforce AWC and are thus omitted, but a well-detailed description of their implementation in GCC can be found in [1]. In general, the purpose of each macro is as follows.

- `va_start` is responsible for populating the initial state of the `va_list`.
- `va_arg` retrieves the next stack argument (of a type specified by the caller).
- `va_copy` makes a clone of the `va_list`.
- and `va_end` performs any necessary maintenance when concluded.

In most C compilers, `va_arg` does not verify that the next argument being accessed belongs to the callee function's variadic argument list; instead, the next region of memory is accessed regardless. Variadic-function attacks are possible because this boundary check is omitted. A function should be restricted to accessing certain data in its own stack frame, but variadic-function attacks can be used to bypass this restriction, defeat Address Space Layout Randomization, and take control of the program by manipulating return addresses [10].

Format-string attacks, a subset of variadic-function attacks that were first discovered in 1999 [2], have garnered a great deal of attention. Formatting functions like `printf` and `fprintf` receive a format string containing a series of format specifiers as input. The format specifiers indicate the location and presentation of data that should be inserted into the format string, and each specifier corresponds to a variadic argument. These functions do not verify their arguments, nor do they ensure that the number of format specifiers matches the number of variadic arguments. Even if an argument is of an incorrect type or does not exist, data will be accessed from the stack and output in accordance with the specifier's rules. An attacker can use these specifiers to leak data and addresses from the stack and, in the case of the `%n` specifier, write arbitrary data to memory [6]. For example, the following C code which prints user output using `printf` can be exploited to leak a return address.

```
void vuln(char *user){ printf(user); }
```

Numerous defenses against format-string attacks have been proposed. One solution is to switch to a type-safe language, but developers often eschew such a time-consuming solution requiring rewriting the codebase. Alternatively, many C and C++ compilers include warnings that alert programs to potentially variadic function vulnerabilities; an example of such a warning is the `-Wformat` GCC option [12]. It is up to the programmer to use these warnings and correct any issues found by the compiler.

More sophisticated defenses have also been developed. An early example is FormatGuard [3], which uses argument counting to abort programs that pass too many arguments to `printf`-like functions. The authors of FormatGuard note that it fails when attackers can achieve their goals without exceeding the passed number of arguments, if the variadic function call is indirect, or if the called function takes a `va_list` directly. These drawbacks yield two failures amongst the tested programs in [3]; format attacks against wu-ftp and gftp still succeed even when protected by FormatGuard. The microbenchmark overhead of FormatGuard was found to be 37%, while the macrobenchmark overhead was 1.3%. Therefore, FormatGuard pays a very low performance cost to mitigate a reasonable portion of format-string attacks.

HexVasan [1] implements type-checking for variadic functions by ensuring that calls to `va_arg` match the real argument types passed to function calls. This strict

type checking violates two rules set by the ISO C standard. First, an object of integer type may be retrieved by specifying either the corresponding integer type or the unsigned integer type, if the value of said integer may be represented in both types. Second, an object of type `void*` may be retrieved by specifying `char*`, and vice versa [6]. HexVasan may also substantially increase the execution time of variadic functions, possibly without significantly affecting the execution time of entire programs. The microbenchmark overhead of HexVasan was found to be 400-600%, while the macrobenchmark overhead was only 0.1-1.2%. Clearly, the macrobenchmarks chosen have a significant impact on the measured overheads, as FormatGuard has a much lower per-variadic-function (microbenchmark) overhead than HexVasan, yet HexVasan reported a much lower per-program (macrobenchmark) overhead. We therefore believe microbenchmarks to more accurately reflect the performance of variadic-function protection mechanisms, but perform both benchmarking strategies to measure our prototype in Section 5.

The LibSafe [18] software package for the GNU C standard library (glibc) intercepts unsafe variadic function calls and replaces them with safe alternatives. The article on LibSafe does not provide performance metrics. Another tool, Lisbon [7], rewrites Win32 binaries to harden them against format-string attacks. Lisbon accomplishes this without needing access to the original source code. However, Lisbon requires debug registers to be available to perform the bounds checking, assumes that variadic arguments are not skipped, and assumes that the `va_list` passed as an argument to vprintf-like functions is located on the stack as part of an upstream caller function. There is also the White-Listing [9] technique, which permits writes only to valid addresses by creating a dynamic range of addresses that may be modified by the `%n` format specifier. Enforcement is then achieved by modifying `printf` to compare the provided address to the list of allowed addresses when using the `%n` format specifier.

It should be noted that the `_FORTIFY_SOURCE` option in GCC does not include protection for variadic arguments. While the fortify optimization does prevent many out-of-bounds issues, the variadic macros are not listed as protected. While the `printf`-like functions are protected, this protection does not extend to the variadic arguments. This can be observed by compiling a simple call such as `printf("%d%d%d")` with the fortify optimization at the highest level. However, this option does provide some protection against format-string attacks by requiring that `%n` specifiers appear in read-only memory (an attack is expected to occur in writable memory space) and disallowing type assumptions for positional arguments in glibc, effectively preventing out of order positional specifiers for attacks.

In summary, a large and diverse array of solutions has been developed to protect software from variadic function vulnerabilities, and each solution has its own strengths and weaknesses. Table 1 summarizes and compares these prior works with the new technique presented here (AWC). Only the incomplete solution of adding compiler warnings has been widely adopted.

Table 1: A comparative analysis of defensive mechanisms against variadic-function attacks

Product	*Attack Type*	*Inject Via*	*Failure Conditions*	*C-Spec Compliant*	*Micro Perf*	*Macro Perf*
FormatGuard	Format str.	Recompilation	`vprintf`-like functions	Yes	37%	<= 1.3%
HexVASAN	Variadic	Recompilation	No known failures	No	400-600%	0.1-1.2%
LibSafe 2.0	Format str.	Lib. Replace	glibc is not used	Yes	Unknown	Unknown
Lisbon	Format str.	Instrumentation	No debug registers	Yes	217.70%	0.3-2%
White-Listing	Format str.	Recompilation	`printf` or `%n` is not used	Yes	10-75%	0.3-1.6%
AWC	Variadic	Recompilation	Mistyping attacks	Yes	17.8-36.1%	0.1-1%

3 Argument Width Counting

Each argument in a function call, when dynamically pushed onto the stack, consumes a predictable number of bytes in memory; the size of this memory space is an argument's width. The callee function should be limited to the allocated region of memory for arguments when accessing the function arguments. By counting the width of variadic arguments that were placed on the stack at the call site, and then tracking the number of bytes consumed by the callee, we can effectively track whether there has been illegal access to stack memory. We refer to this memory access-control policy as Argument Width Counting (AWC). AWC is enforceable because the width of the arguments can be reliably calculated at runtime, and invocations to `va_arg` provide the number of bytes being accessed.

When a function call begins, each argument is pushed to the stack in reverse order, as shown in Figure 1. Before the last variadic argument is pushed onto the stack (i.e., `ESP` in Figure 1 points to the entry marked "Saved EBP"), AWC records the value of the stack pointer, and after the first variadic argument is pushed (i.e., `ESP` points to the entry marked "Variadic Argument 1"), AWC subtracts the current stack pointer from the recorded value. The result of this subtraction gives AWC the sum of the widths of all variadic arguments. This total width can then be tracked and decremented each time `va_arg` is called, and an error can be emitted if the total width is bypassed.

As an optimization, it is often unnecessary to perform the subtraction dynamically (to compute the total width of function arguments), because the widths of the arguments are likely to be known statically by the compiler. In this case, we can store the statically known total width of arguments at compile time; our prototype implementation takes advantage of this optimization.

The exact mechanism implemented to enforce the AWC policy may vary, but can generally be split into two parts: 1) storing the remaining width, and 2) validating the

va_arg call during execution. One potential storage mechanism may be similar to HexVASAN's, where the recorded width sums are kept in a thread-local map keyed by va_list pointers, placing the sum into the map via va_start, and decrementing the sum via va_arg. Depending on the level of control the compiler has, a new tracking field may instead be added directly to the va_list struct at compile time. We use this technique in our prototype mechanism due to the performance advantages. This optimization is possible as the va_list struct is not well defined by the C standard, and thus is implementation dependent [14, 17]. Fortunately, the macros that manage va_list are also implementation dependent, and validation checks can be added to abort the program if va_arg is invoked invalidly without breaking from the C specification, as it is undefined behavior [17]. Our implementation emits these changes during compilation, but other implementations could, for example, be developed to add them to an existing binary.

AWC is capable of mitigating all known attacks in which the attacker uses va_arg to escape the stack space of variadic arguments to gain access to other program memory on the stack, which composes the vast majority of observed attacks against variadic functions. AWC is however less comprehensive than HexVASAN, as it does not enforce that the arguments are of a specific type, but this limitation of AWC may not be of great practical significance because previous work has observed no attacks that are based strictly on mistyping variadic arguments [3]. Additionally, AWC's lack of strict type enforcement allows for full compliance with the C standard, in contrast to HexVASAN (as explained in the preceding paragraph). Finally, AWC's simplified mechanism enables it to achieve performance overheads superior to HexVASAN.

Unlike FormatGuard, which counts and validates the number of arguments provided to the printf function, AWC's more generalized approach can be applied to all uses of the va_arg function, which provides additional security to all programs that use variadic functions, not just those that use printf. AWC's protection encompasses other features that FormatGuard could not protect, such as indirect calls and calls to vsprintf (or similar functions) that take a va_list directly. Our AWC implementation accomplishes this by storing the data in the va_list struct, allowing the verification to be performed wherever va_arg is called; other implementations such as those mentioned earlier can achieve similar effects. Despite this additional protection, AWC achieves low overhead comparable to FormatGuard, discussed further in Section 5.

We therefore believe that AWC strikes a good balance between security and performance while complying with the C standard.

4 Efficiently Adding AWC to Compilers

A prototype AWC mechanism was implemented using Clang, a C/C++ compiler built upon the LLVM code generator and optimizer [15]. The implementation is

publicly available on GitHub, including a detailed write-up, build scripts, a comprehensive set of test cases, and benchmarking programs [4, 5].

While AWC can be achieved in a variety of ways, as discussed in Section 3, we believe the simplest and most efficient method is to add AWC during compilation. In order to accommodate AWC, a compiler must satisfy four requirements. In the unlikely case that any one criterion is not satisfied, AWC will be difficult or impossible to fully implement using the techniques described in this section and another technique would be needed instead. The four requirements are:

1. an entry point for function call code generation;
2. the ability to determine if a function call is variadic or not;
3. knowing the number of bytes that function parameters take up on the stack;
4. and the ability to modify the variadic macros and `va_list` struct.

Clang meets all of these requirements and all of the necessary modifications are available and described in detail on Github [4]. The authors are unaware of any C/C++ compiler that would be unable to meet these requirements. While some compilers may need larger modifications than others, all of the required information and features would be necessary to fully implement a C compiler that meets the C standard. GCC's ability to meet these requirements is described in more detail in Section 6.

4.1 Implementing AWC in Clang

When a function call is generated, our modified Clang implementation will now inject the variadic argument width into the argument list between the last non-variadic argument and the first variadic argument. This can be done simply by tracking the number of arguments that have been generated; if there are N non-variadic arguments, then the width should be generated as an argument after N arguments have been generated. While Clang does determine if a function is variadic or not, it does not directly calculate the number of non-variadic and variadic arguments. However, Clang does need to iterate over every argument to determine and validate the argument type. Non-variadic argument types are checked first, as they come directly from the function declaration. All of the remaining arguments are variadic and their types are determined afterwards based on the argument expression type. As the types are stored in a vector, the number of non-variadic arguments can be determined by checking the size of the vector between these two steps. Finally, the total width of the variadic arguments can also be determined while iterating over their types by summing the memory size of each type. The addition to Clang to accomplish this is shown in Figure 2; note that the width of each argument is shown in bits, so the value is divided by 8 to convert to bytes.

With the argument width injected, the next step is locating it on the stack when `va_start` is called. Clang (or, more precisely, the LLVM x86-64 backend) does

```
size_t numNonVariadic = ArgTypes.size();
uint64_t varArgsSize = 0;
for(auto *A:llvm::drop_begin(ArgRange,ArgTypes.size()))
{
    ...
    varArgsSize += getContext().getTypeSize(argType)/8;
}
```

Fig. 2: Calculating the sum of widths during a function call in Clang

not use the traditional cdecl call style discussed earlier. Instead, a register-first calling convention is implemented, which places some arguments into registers rather than on the stack. Note that this calling convention and the terminology are not as standardized as cdecl; while the high-level implementation will be similar, there may be variations depending on the compiler and operating system (e.g., number of registers and the assumptions about the contents of those registers). Clang places the first 6 integral arguments and the first 6 floating pointing arguments in registers. The variadic function implementation is designed to account for this by copying the registers onto the stack when the variadic function begins. This means that there are now two locations where the variadic width can be found: either in the register store area copied onto the stack or in the normal stack location above the return address where they are normally found.

To determine which location the width has been stored, it is necessary to understand how `va_arg` retrieves the next variadic argument. The layout for Clang and the relevant `va_list` values are visualized in Figure 3. The `va_list` struct contains the `reg_save_area` pointer, which is the address of the first register argument stored on the stack. The five arguments (or a 0 if the argument is non-variadic) are stored between `reg_save_area` and `reg_save_area+40`. When `va_start` is executed, `gp_offset` in the `va_list` is initialized to the corresponding offset for the first entry with a variadic argument, or 48 if all six arguments were non-variadic. Thus, each call to `va_arg` retrieves `reg_save_area + gp_offset` and then increments `gp_offset` by 8. Once `gp_offset` is 48, all of the variadic arguments in the register area have been consumed, and `overflow_arg_area`, which points to the next variadic argument on the stack, is retrieved and incremented instead. Note that the same process is done for the floating-point arguments that were stored in registers, using `fp_offset` as the offset instead.

Our Clang implementation must perform this lookup when running `va_start`. In the worst-case scenario, it is equivalent to running `va_arg` an additional time. Some optimizations can be made, however, since the width will always be the first variadic argument. In Clang's implementation, the calculation could be entirely performed at compile time, since calls to `va_start` are entirely inlined and the location could be determined based on the number of non-variadic arguments passed

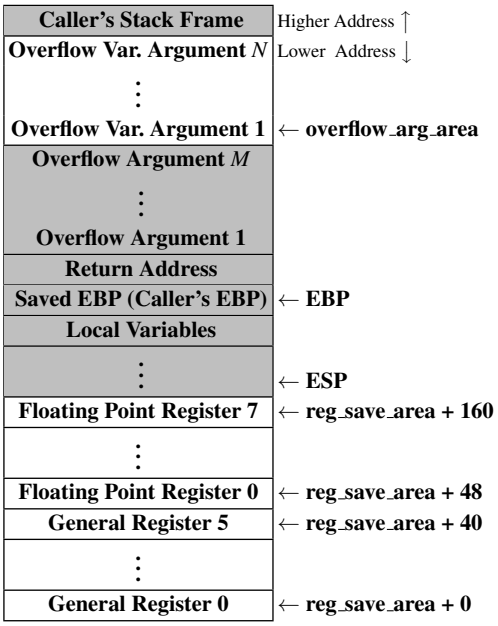

Fig. 3: Layout of the x86 stack with Clang during a variadic function call

to the function (i.e., if six non-variadic arguments are passed, then use the overflow pointer rather than the register save area). However, as this code is generated as part of LLVM's x86 backend implementation for `va_start`, access to the function prototype was not previously necessary and is not available. An additional argument could be added to LLVM's code generation call for `va_start` with the number of non-variadic arguments, but this would require changing LLVM's lowering structure. This is undesirable for our prototype, as the intention is to limit large changes to Clang/LLVM, so instead our prototype generates a call to `va_arg` at the end of `va_start` to retrieve the value.

The final modification is to check and decrement the remaining width during `va_arg`. This is achieved by taking advantage of the Clang code builder API, which builds an execution chain processed by LLVM. The binary `AND` operation with the value `0x8000000000000000` is equivalent to `remaining < 0` but is used instead of a standard comparison as this operation can be executed faster.

5 Empirical Analysis of AWC

The performance cost of AWC was evaluated by conducting a microbenchmark of the Clang compiler; that is, a stress test was performed that focused specifically on the execution of the impacted variadic functions. In addition, Clang was mac-

robenchmarked on the man2html program [11] as this was used by existing work for benchmarking. The following subsections present the benchmarking processes and the results, comparing the standard compilers to their AWC-enabled alternatives. All of the testing files and scripts are available in the public GitHub repository [5].

5.1 Microbenchmarking AWC

A test program that stresses the variadic functions was developed to benchmark performance with and without the addition of AWC. The benchmark program, with the relevant function shown in Figure 4, takes a number of integers as variadic arguments and returns the sum of those integers. The number of arguments to be summed, *n*, is passed as the first argument, followed by all of the integers to be summed as variadic arguments. This function has one invocation of `va_start` and `va_end`, and *n* invocations of `va_arg` (where the relatively expensive check is performed). The function is concise, providing enough complexity to evaluate the overhead of the variadic arguments, but without being overshadowed by other potentially expensive or volatile computations (e.g., the output of a function like `printf`).

```
int sumN(int n, ...){
  va_list args; int sum = 0;
  va_start(args, n);
  for(int i = 0; i < n; i++)
    sum += va_arg(args, int);
  return sum;
}
```

Fig. 4: A variadic summation function used for benchmarking AWC-enabled compilers

The C standard function `clock()` was used to measure the runtime of the `sumN` function. A small wrapper function records the start time, performs one million invocations of `sumN()`, and then concludes by recording the final end time. The CPU time consumed was calculated by dividing the difference between the end and start times by the C standard `CLOCKS_PER_SEC` constant. The benchmarking was performed with an increasing number of arguments to determine how the performance scaled with the number of arguments provided to a variadic function. The benchmark program was run 1000 times per case, and the final recorded time was the average of those runs. Thus, each of the 1000 runs yielded the cost of one million invocations of `sumN`; a total of 1 billion invocations for each of the given number of arguments. For all cases, the system was a Ubuntu 22.04.3 LTS desktop (under the Windows Subsystem for Linux on Windows 11) utilizing an Intel i7-1355U pro-

cessor and 32 GB of RAM. The results of the benchmark for Clang are recorded in Table 2.

Table 2: Microbenchmark results for Clang with AWC

# of Args	Time (ms)	Time w/ AWC	Overhead (ms)	Overhead (%)
0	2.435	2.993	0.558	22.935
3	5.610	6.816	1.206	21.507
16	28.886	34.024	5.137	17.785
100	117.971	160.528	42.557	36.074

These benchmark results show the performance impacts of AWC on a simple variadic function. When provided between 0-16 arguments, AWC performs at about a fairly consistent 20% overhead, before scaling up to 36% at 100 arguments. We believe this scaling is due to the different calculations that occur based on the location of the next variadic argument to retrieve. The first 6 arguments all require additional calculations to locate their position, while arguments beyond that are located immediately by a direct pointer. The AWC-enabled compiler, however, has an extra calculation to perform regardless, and this accumulates in a noticeable performance difference when scaled up to a large number of variadic arguments. Thus, AWC exhibits a nonlinear time increase with respect to the number of variadic arguments.

5.2 Macrobenchmarking Clang with AWC

In addition to the microbenchmarking, the modified Clang compiler was also benchmarked using a real-world program, man2html. The man2html program, which converts Linux man pages into an HTML document, was chosen as it makes frequent use of variadic functions (`printf` and `printf`-like functions), is modified infrequently, and was also used for benchmarking by two of the existing mechanisms described in Section 2.

Man2html was compiled twice, once with the original Clang compiler, and once with the modified, AWC-enabled version. The man2html program was then run on four man pages available by default on Ubuntu 22, chosen by size: `rbash` at 161 bytes (smallest), `modify_ldt` at 5636 bytes (median), `dpkg-buildpackage` at 27,782 bytes (mean), and `x86_64-linux-gnu-gcc` at 1,407,155 bytes (largest). Each case was run 10,000,000 times, and the average time taken was calculated. The results of the benchmarking can be found in Table 3.

Reviewing the results, the performance differences in man2html compiled with and without AWC are insignificant, just below 1% in the worst-performing case. Such differences are only noticeable when averaged over such a large scale and are indistinguishable from environmental factors on an individual basis; smaller-scale tests will regularly show AWC outperforming the non-AWC version.

Table 3: Benchmarking Clang with AWC on man2html

Program	Time (ms)	Time w/ AWC	Overhead (ms)	Overhead (%)
rbash	1.367	1.378	0.011	0.830
modify_ldt	1.483	1.485	0.002	0.121
dpkg	1.909	1.928	0.019	0.984
gcc	29.532	29.673	0.140	0.473

6 Discussion

Enforcing AWC mitigates all observed variadic-function attacks without limiting or otherwise invalidating any intended functionality. AWC's protection extends to `va_copy`, passing `va_list` variables as function arguments, and complex use cases such as those seen in `printf`. This protection is extended to the positional parameters supported in glibc as these function by repeatedly calling `va_arg` [5]. Protection is not extended to the `va_list` structs themselves; programs that access and manipulate the underlying data structures directly could still be exploited. However, such code would need to be specific to the compiler implementation and would violate the defined standard.

AWC can be easily implemented in a compiler without the need to modify the source program. There are no unintended side effects from enforcing AWC with our prototype implementation; all C programs that avoid undefined behavior according to the C standard specification will be compiled into an output program that is functionally equivalent to the output program without AWC protections (excluding, of course, the intended termination of the program when making an out-of-bounds `va_arg` call). Programs that violate the C standard (e.g., by making compiler-specific assumptions) will function equivalently as long as `va_start` is used to initialize all `va_list` variables, as this will account for the width argument being stored as the first variadic argument; even manually implementing `va_arg` will work equivalently, since the necessary calculations for `va_arg` are unchanged after `va_start` is called.

The AWC policy is quite flexible as the assumptions or requirements to enforce the policy are minor. For example, moving the variadic arguments or the `va_list` from the stack to the heap or to a different location on the stack would require, at most, trivial changes to the AWC implementation. This flexibility is possible as only existing compiler features and variadic macros are used to implement the runtime monitor (specifically, the existing function to add variadic arguments and `va_arg`). Some of the proposed runtime improvements may make such assumptions about the layout, but the current prototype makes no such assumptions.

The prototype AWC mechanism also shows promising performance results when compared to prior solutions. FormatGuard imposes a consistent 37% overhead on calls to `printf`, which is higher than the cost incurred by AWC, though it does not scale with the number of arguments [3]. Interestingly, FormatGuard was re-benchmarked and shown to have argument-scaling performance costs on `sprintf`,

hitting 38% at just two `%n` specifiers (versus 7.5% with no arguments) [7]. HexVASAN incurs 4-6 times the overhead on variadic function calls, which is an order of magnitude higher than AWC's performance impact [1]. Libsafe does not provide any performance metrics, so it is impossible to compare the cost of that implementation with AWC [18]. Lisbon shows performance costs in the 1.5 to 3 times range, incurring the highest percentage overhead when zero arguments are present. White-Listing incurred a performance cost ranging from 10% to 75%, based on the specific function being called, the number of arguments, and the type of format specifiers present.

The AWC-enabled Clang prototype outperforms, or is on par, with each of these prior implementations. With AWC achieving a microbenchmarked overhead of around 20-30%, only White-Listing outperforms AWC in some scenarios. White-Listing, however, only applies to `printf`-like functions and only protects against attacks that use `%n` modifiers. When macrobenchmarked on the man2html program, AWC achieves an overhead between 0.1% to 0.98%, again performing on par or better than the other solutions. AWC still has potential room for improvement as well. By performing the `va_arg` calculation at compile time, the performance can be improved even further as the added instructions are reduced down to three store/load instructions, with no branching conditionals necessary. While the modifications were not fully tested, as they would require changes to LLVM's lowering structure API for the variadic functions, a preliminary version achieved less than 5% overhead when passing 0 variadic arguments in the microbenchmark program.

We first investigated the possibility of adding AWC to GCC (instead of Clang) using the same technique described in Section 4. While GCC does, strictly speaking, meet the four requirements, the variadic macros and `va_list` struct are written in a builtin and stored directly as pre-generated intermediate code, rather than C/C++ code. An engineer wishing to edit them would need to identify their locations in the code and understand the machine-independent language called Register Transfer Language Expressions (RTX). While RTX is documented, GCC considers the variadic macros internal. This designation means that GCC provides no documentation on them, their operations, or their locations [13]. Modification of these macros requires extensive knowledge of GCC and is beyond the scope of this work. For this reason, Clang was instead chosen as the various variadic functions are implemented in C++ as part of the LLVM backend. As the change would consist of only a single conditional check, we speculate that a software engineer with some familiarity with GCC and RTX could quickly implement this step.

6.1 Bypassing AWC Protection

As with other safeguards added to the C programming language, AWC is not infallible. There are two situations where the authors foresee an attacker potentially bypassing the protection from AWC: corrupting the `remaining` value in the `va_list` or mistyping the variadic arguments. Fortunately, the authors predict sit-

uations where these are likely to occur and also be of value to an attacker are rare, as discussed below.

The first attack is straightforward; if an attacker can modify the `remaining` value, then they can increase the value to allow for out-of-bounds reads by causing `va_arg` to be called too many times. However, this modification must occur between the call to `va_start` and the malicious `va_arg` call; the `remaining` value will not be available to modify before `va_start`, and the program will terminate if the malicious `va_arg` call is made before `remaining` is increased. This means such an attack is likely impossible against `printf` unless some new vulnerability is discovered in the `printf` implementation that allows for `remaining` to be modified.

One major complication with using a buffer overflow to perform such an attack is that `remaining` is located at the end of the `va_list` (putting it at a higher address on the stack) and requires the attacker to overwrite all of the other values in the struct before overwriting `remaining`. This means that the attacker must replace these values or the `va_arg` call will segfault and crash the program; thus, this requires the attacker to leak the stack addresses to perform this attack. Considering a typical use case for a variadic function attack is to leak stack addresses to use in another attack, such an attack might not grant the attacker any new capabilities.

Finally, properly ordering local variables on the stack so buffer overflows cannot overwrite non-buffer local variables (a technique automatically implemented by default during compilation with GCC) greatly reduces the threat of such an attack occurring as well. More complicated techniques that allow for arbitrary writes to memory would also succeed in bypassing AWC, but if an attacker can successfully perform such an attack, they most likely do not even need to manipulate the variadic arguments to achieve their goals.

The second attack, where a variadic argument is mistyped, was previously described in Section 2. AWC is not designed to prevent such an attack. In addition to the reasons described in the previous section (i.e., lack of natural occurrence and compliance with the C standard), this attack is not a consequence of the variadic argument implementation and is instead due to C lacking type checking at runtime. While the most logical place to find such mistyping attacks is in combination with a variadic function attack, they are possible in other scenarios and are not caused by the variadic functions. These mistyping attacks will be possible anywhere an attacker can control a variable's content and the type it is processed as. Such attacks are unlikely to be observed naturally, however, as they require functions to be used in contrived ways, such as `printf(user_input, user_input2)`, where the user input supplies both the format string and the first argument.

7 Conclusions

Variadic-function attacks (and by extension, format-string attacks) can be just as severe as buffer overflow attacks and present an ongoing threat to software ap-

plications. Despite numerous proposed mitigations, ranging from simple argument counting via wrapper functions to a full type-checking system for variadic functions, widespread adoption has remained elusive. AWC is a new solution for mitigating variadic function attacks by ensuring variadic functions cannot misuse invalid memory as function arguments. AWC is effective, low cost, and can be applied to compilers, providing universal protection across all generated programs. Once applied to a compiler, all emitted programs will benefit from variadic attack mitigations. The empirical evidence presented in Section 5 shows AWC is at least as effective as the best-known real-world attack mitigation solution, HexVASAN, and manages to outperform the most lightweight solutions like FormatGuard and White-Listing.

This significant performance uplift is achieved by leveraging the fact that the C specification does not define how a program should access variadic arguments, allowing for changes to be made without becoming incompliant with the spec. Prior attempts have not made direct changes to this mechanism, instead applying modifications to usage sites, if at all [1,3,7,9,18]. The change made by AWC does change the size of the compiler-internal struct `va_list` and changes the operations done by the compiler-internal macros `va_start`, `va_arg`, and `va_end`, which could potentially be a problem if a program was relying on implementation details. However, these are builtins, and as such are not public API, so programs that are relying on implementation details of these builtins are doing so at their own risk, as they could change at any time [13].

Future work could prioritize the integration of AWC into other prominent optimizing compilers like GCC, including its inclusion in future upstream releases. Fine-tuning is essential, especially to enable mixed AWC and non-AWC program execution, but this level of compatibility could be achieved with minimal difficulty at the cost of introducing some security considerations for non-AWC-enabled programs. Alternatively, future work could focus on determining if the mechanism utilized here is the most efficient for enforcing AWC. Such alternative mechanisms may differ in how they calculate or pass the argument width through the call site, track the consumption of arguments, or optimize the check done from within `va_arg` (e.g., implementing directly in assembly).

References

1. Biswas, P., Federico, A.D., Carr, S.A., Rajasekaran, P., Volckaert, S., Na, Y., Franz, M., Payer, M.: Venerable variadic vulnerabilities vanquished. In: 26th USENIX Security Symposium (USENIX Security 17), pp. 186–198. USENIX Association, Vancouver, BC (2017). URL `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/biswas`
2. Common Weakness Enumeration: Cwe-134: Use of externally-controlled format string. `https://cwe.mitre.org/data/definitions/134.html` (2023). Retrieved September 7, 2023
3. Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G., Frantzen, M., Lokier, J.: FormatGuard: Automatic protection from printf format string vulnerabilities. In: 10th USENIX Security Symposium (USENIX Security 01).

USENIX Association, Washington, D.C. (2001). URL https://www.usenix.org/conference/10th-usenix-security-symposium/formatguard-automatic-protection-printf-format-string
4. Dennis, K.: argwidthcounting/awc-clang: Clang compiler with AWC. https://github.com/Ktrio3/awc-clang (2024). Retrieved January 29, 2024
5. Dennis, K.: argwidthcounting/awc-test-suite: Testing suite for AWC. https://github.com/Ktrio3/awc-test-suite (2024). Retrieved January 29, 2024
6. International Standards Organization: ISO/IEC:9899:TC2 (programming languages - C) (2005). URL \url{https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf}. Retrieved September 9, 2023
7. Li, W., Chiueh, T.c.: Automated format string attack prevention for win32/x86 binaries. In: Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), pp. 398–409 (2007). DOI 10.1109/ACSAC.2007.23
8. Microsoft: _cdecl. https://learn.microsoft.com/en-us/cpp/cpp/cdecl?view=msvc-170 (2021). Retrieved September 9, 2023
9. Ringenburg, M.F., Grossman, D.: Preventing format-string attacks via automatic and efficient dynamic checking. In: Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS '05, p. 354–363. Association for Computing Machinery, New York, NY, USA (2005). DOI 10.1145/1102120.1102166. URL https://doi.org/10.1145/1102120.1102166
10. Roemer, R., Buchanan, E., Shacham, H., Savage, S.: Return-oriented programming: Systems, languages, and applications. ACM Trans. Inf. Syst. Secur. **15**(1) (2012). DOI 10.1145/2133375.2133377. URL https://doi.org/10.1145/2133375.2133377
11. snapshot.debian.org: man2html - debian snapshot. https://snapshot.debian.org/archive/debian/20110109T212222Z/pool/main/m/man2html (2024). Retrieved January 29, 2024
12. The GNU Project: 3.8 options to request or suppress warnings. https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html (2023). Retrieved September 9, 2023
13. The GNU Project: 6.59 other built-in functions provided by gcc. https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html (2023). Retrieved September 10, 2023
14. The GNU Project: A.2 variadic functions. https://www.gnu.org/software/libc/manual/html_node/Variadic-Functions.html (2023). Retrieved September 10, 2023
15. The LLVM Organization: The llvm compiler infrastructure. https://github.com/llvm/llvm-project (2024). Retrieved January 29, 2024
16. The MITRE Corporation: Cve search results for 'format string'. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=format%20string (2023). Retrieved September 10, 2023
17. The Open Group: stdarg.h. https://pubs.opengroup.org/onlinepubs/007904975/basedefs/stdarg.h.html (2021). Retrieved September 10, 2023
18. Tsai, T., Singh, N.: Libsafe 2.0: Detection of format string vulnerability exploits. Tech. rep., Avaya Labs, Murray Hill, NJ (2001)