Venerable Variadic Vulnerabilities Vanquished

Priyam Biswas[1], Alessandro Di Federico[2], Scott A. Carr[1], Prabhu Rajasekaran[3], Stijn Volckaert[3], Yeoul Na[3], Michael Franz[3], and Mathias Payer[1]

[1]Department of Computer Science, Purdue University
{biswas12, carr27}@purdue.edu, mathias.payer@nebelwelt.net
[2]Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano
alessandro.difederico@polimi.it
[3]Department of Computer Science, University of California, Irvine
{rajasekp, stijnv, yeouln, franz}@uci.edu

Abstract

Programming languages such as C and C++ support variadic functions, i.e., functions that accept a variable number of arguments (e.g., `printf`). While variadic functions are flexible, they are inherently not type-safe. In fact, the semantics and parameters of variadic functions are defined implicitly by their implementation. It is left to the programmer to ensure that the caller and callee follow this implicit specification, without the help of a static type checker. An adversary can take advantage of a mismatch between the argument types used by the caller of a variadic function and the types expected by the callee to violate the language semantics and to tamper with memory. Format string attacks are the most popular example of such a mismatch.

Indirect function calls can be exploited by an adversary to divert execution through illegal paths. CFI restricts call targets according to the function prototype which, for variadic functions, does not include all the actual parameters. However, as shown by our case study, current CFI implementations are mainly limited to non-variadic functions and fail to address this potential attack vector. Defending against such an attack requires a stateful dynamic check.

We present HexVASAN, a compiler based sanitizer to effectively type-check and thus prevent any attack via variadic functions (when called directly or indirectly). The key idea is to record metadata at the call site and verify parameters and their types at the callee whenever they are used at runtime. Our evaluation shows that HexVASAN is (i) practically deployable as the measured overhead is negligible (0.45%) and (ii) effective as we show in several case studies.

1 Introduction

C and C++ are popular languages in systems programming. This is mainly due to their low overhead abstractions and high degree of control left to the developer. However, these languages guarantee neither type nor memory safety, and bugs may lead to memory corruption. Memory corruption attacks allow adversaries to take control of vulnerable applications or to extract sensitive information.

Modern operating systems and compilers implement several defense mechanisms to combat memory corruption attacks. The most prominent defenses are Address Space Layout Randomization (ASLR) [47], stack canaries [13], and Data Execution Prevention (DEP) [48]. While these defenses raise the bar against exploitation, sophisticated attacks are still feasible. In fact, even the combination of these defenses can be circumvented through information leakage and code-reuse attacks.

Stronger defense mechanisms such as Control Flow Integrity (CFI) [6], protect applications by restricting their control flow to a predetermined control-flow graph (CFG). While CFI allows the adversary to corrupt non-control data, it will terminate the process whenever the control-flow deviates from the predetermined CFG. The strength of any CFI scheme hinges on its ability to statically create a precise CFG for indirect control-flow edges (e.g., calls through function pointers in C or virtual calls in C++). Due to ambiguity and imprecision in the analysis, CFI restricts adversaries to an over-approximation of the possible targets of individual indirect call sites.

We present a new attack against widely deployed mitigations through a frequently used feature in C/C++ that has so far been overlooked: variadic functions. Variadic functions (such as `printf`) accept a varying number of arguments with varying argument types. To implement variadic functions, the programmer implicitly encodes the argument list in the semantics of the function and has to make sure the caller and callee adhere to this implicit contract. In `printf`, the expected number of arguments and their types are encoded implicitly in the format string, the first argument to the function. Another frequently used scheme iterates through parameters until a condition is reached (e.g., a parameter is NULL). Listing 1 shows an example of a variadic function. If an adversary can violate the implicit contract between caller and callee, an attack may be possible.

In the general case, it is impossible to enumerate the arguments of a variadic function through static analysis techniques. In fact, their number and types are intrinsic in how the function is defined. This limitation enables (or facilitates) two attack vectors against variadic functions. First, attackers can hijack indirect calls and thereby call variadic functions over control-flow edges that are never taken during any legitimate execution of the program. Variadic functions that are called in this way may interpret the variadic arguments differently than the function for which these arguments were intended, and thus violate the implicit caller-callee contract. CFI countermeasures specifically prevent illegal calls over indirect call edges. However, even the most precise implementations of CFI, which verify the type signature of the targets of indirect calls, are unable to fully stop illegal calls to variadic functions.

A second attack vector involves overwriting a variadic function's arguments directly. Such attacks do not violate the intended control flow of a program and thus bypass all of the widely deployed defense mechanisms. Format string attacks are a prime example of such attacks. If an adversary can control the format string passed to, e.g., printf, she can control how all of the following parameters are interpreted, and can potentially leak information from the stack, or read/write to arbitrary memory locations.

The attack surface exposed by variadic functions is significant. We analyzed popular software packages, such as Firefox, Chromium, Apache, CPython, nginx, OpenSSL, Wireshark, the SPEC CPU2006 benchmarks, and the FreeBSD base system, and found that variadic functions are ubiquitous. We also found that many of the variadic function calls in these packages are indirect. We therefore conclude that both attack vectors are realistic threats. The underlying problem that enables attacks on variadic functions is the lack of type checking. Variadic functions generally do not (and cannot) verify that the number and type of arguments they expect matches the number and type of arguments passed by the caller. We present HexVASAN, a compiler-based, dynamic sanitizer that tackles this problem by enforcing type checks for variadic functions at run-time. Each argument that is retrieved in a variadic function is type checked, enforcing a strict contract between caller and callee so that (i) a maximum of the passed arguments can be retrieved and (ii) the type of the arguments used at the callee are compatible with the types passed by the caller. Our mechanism can be used in two operation modes: as a runtime monitor to protect programs against attacks and as sanitizer to detect type mismatches during program testing.

We have implemented HexVASAN on top of the LLVM compiler framework, instrumenting the compiled code to record the types of each argument of a variadic function at the call site and to check the types whenever they are retrieved. Our prototype implementation is light-weight, resulting in negligible (0.45%) overhead for SPEC CPU2006. Our approach is general as we show by recompiling the FreeBSD base system and effective as shown through several exploit case studies (e.g., a format string vulnerability in sudo).

We present the following contributions:

- Design and implementation of a variadic function sanitizer on top of LLVM;
- A case study on large programs to show the prevalence of direct and indirect calls to variadic functions;
- Several exploit case studies and CFI bypasses using variadic functions.

2 Background

Variadic functions are used ubiquitously in C/C++ programs. Here we introduce details about their use and implementation on current systems, the attack surface they provide, and how adversaries can abuse them.

```
#include <stdio.h>
#include <stdarg.h>

int add(int start, ...) {
    int next, total = start;
    va_list list;
    va_start(list, start);
    do {
        next = va_arg(list, int);
        total += next;
    } while (next != 0);
    va_end(list);
    return total;
}

int main(int argc, const char *argv[]) {
    printf("%d\n", add(5, 1, 2, 0));
    return 0;
}
```

Listing 1: Example of a variadic function in C. The function add takes a non-variadic argument start (to initialize an accumulator variable) and a series of variadic int arguments that are added until the terminator value 0 is met. The final value is returned.

2.1 Variadic functions

Variadic functions (such as the printf function in the C standard library) are used in C to maximize the flexibility in the interface of a function, allowing it to accept a number of arguments unknown at compile-time. These functions accept a variable number of arguments, which do not necessarily have fixed types. An example of a variadic function is shown in Listing 1. The function add accepts one mandatory argument (start) and a varying number of additional arguments, which are marked by the ellipsis (...) in the function definition.

The C standard defines several macros that portable programs may use to access variadic arguments [33]. stdarg.h, the header that declares these macros, defines an opaque type, va_list, which stores all information required to retrieve and iterate through variadic arguments. In our example, the variable list of type va_list is initialized using the va_start macro. The va_arg macro retrieves the next variadic argument from the va_list, updating va_list to point to the next argument as a side effect. Note that, although the programmer must specify the expected type of the variadic argument in the call to va_arg, the C standard does not require the compiler to verify that the retrieved variable is indeed of that type. va_list variables must be released using a call to the va_end macro so that all of the resources assigned to the list are deallocated.

printf is an example of a more complex variadic function which takes a format string as its first argument. This format string implicitly encodes information about the number of arguments and their type. Implementations of printf scan through this format string several times to identify all format arguments and to recover the necessary space in the output string for the specified types and formats. Interestingly, arguments do not have to be encoded sequentially but format strings allow out-of-order access to arbitrary arguments. This flexibility is often abused in format string attacks to access arbitrary stack locations.

2.2 Variadic functions ABI

The C standard does not define the calling convention for variadic functions, nor the exact representation of the va_list structure. This information is instead part of the ABI of the target platform.

x86-64 ABI. The AMD64 System V ABI [36], which is implemented by x86-64 GNU/Linux platforms, dictates that the caller of a variadic function must adhere to the normal calling conventions when passing arguments. Specifically, the first six non-floating point arguments and the first eight floating point arguments are passed through CPU registers. The remaining arguments, if any, are passed on the stack. If a variadic function accepts five mandatory arguments and a variable number of variadic arguments, then all but one of these variadic arguments will be passed on the stack. The variadic function itself moves the arguments into a va_list variable using the va_start macro. The va_list type is defined as follows:

```
typedef struct {
  unsigned int gp_offset;
  unsigned int fp_offset;
  void *overflow_arg_area;
  void *reg_save_area;
} va_list[1];
```

va_start allocates on the stack a reg_save_area to store copies of all variadic arguments that were passed in registers. va_start initializes the overflow_arg_area field to point to the first variadic argument that was passed on the stack. The gp_offset and fp_offset fields are the offsets into the reg_save_area. They represent the first unused variadic argument that was passed in a general purpose register or floating point register respectively.

The va_arg macro retrieves the first unused variadic argument from either the reg_save_area or the overflow_arg_area, and either it increases the gp_offset/fp_offset field or moves the overflow_arg_area pointer forward, to point to the next variadic argument.

Other architectures. Other architectures may implement variadic functions differently. On 32-bit x86, for example, all variadic arguments must be passed on the stack (pushed right to left), following the cdecl calling convention used on GNU/Linux. The variadic function itself retrieves the first unused variadic argument directly from the stack. This simplifies the implementation of the va_start, va_arg, and va_end macros, but it generally makes it easier for adversaries to overwrite the variadic arguments.

2.3 Variadic attack surface

When calling a variadic function, the compiler statically type checks all non-variadic arguments but does not enforce any restriction on the type or number of variadic arguments. The programmer must follow the implicit contract between caller and callee that is only present in the code but never enforced explicitly. Due to this high flexibility, the compiler cannot check arguments statically. This lack of safety can lead to bugs where an adversary achieves control over the callee by modifying the arguments, thereby influencing the interpretation of the passed variadic arguments.

Modifying the argument or arguments that control the interpretation of variadic arguments allows an adversary to change the behavior of the variadic function, causing the callee to access additional or fewer arguments than specified and to change the interpretation of their types.

An adversary can influence variadic functions in several ways. First, if the programmer forgot to validate the input, the adversary may directly control the arguments to the variadic function that controls the interpretation of arguments. Second, the adversary may use an arbitrary memory corruption elsewhere in the program to influence the argument of a variadic function.

Variadic functions can be called statically or dynamically. Direct calls would, in theory, allow some static checking. Indirect calls (e.g., through a function pointer), where the target of the variadic function is not known, do not allow any static checking. Therefore, variadic functions can only be protected through some form of runtime checker that considers the constraints of the call site and enforces them at the callee side.

2.4 Format string exploits

Format string exploits are a perfect example of corrupted variadic functions. An adversary that gains control over the format string used in `printf` can abuse the `printf` function to leak arbitrary data on the stack or even resort to arbitrary memory corruption (if the pointer to the target location is on the stack). For example, a format string vulnerability in the smbclient utility (CVE-2009-1886) [40] allows an attacker to gain control over the Samba file system by treating a filename as format string. Also, in PHP 7.x before 7.0.1, an error handling function in zend_execute_API.c allows an attacker to execute arbitrary code by using format string specifiers as class name (CVE-2015-8617) [1].

Information leaks are simple: an adversary changes the format string to print the desired information that resides somewhere higher up on the stack by employing the desired format string specifiers. For arbitrary memory modification, an adversary must have the target address encoded somewhere on the stack and then reference the target through the `%n` modifier, writing the number of already written bytes to that memory location.

The GNU C standard library (*glibc*) enforces some protection against format string attacks by checking if a format string is in a writable memory area [29]. For format strings, the *glibc* `printf` implementation opens `/proc/self/maps` and scans for the memory area of the format string to verify correct permissions. Moreover, a check is performed to ensure that all arguments are consumed, so that no out-of-context stack slots can be used in the format string exploit. These defenses stop some attacks but do not mitigate the underlying problem that an adversary can gain control over the format string. Note that this heavyweight check is only used if the format string argument *may* point to a writable memory area at compile time. An attacker may use memory corruption to redirect the format string pointer to an attacker-controlled area and fall back to a regular format string exploit.

3 Threat model

Programs frequently use variadic functions, either in the program itself or as part of a shared library (e.g., `printf` in the C standard library). We assume that the program contains an arbitrary memory corruption, allowing the adversary to modify the arguments to a variadic function and/or the target of an indirect function call, targeting a variadic function.

Our target system deploys existing defense mechanisms like DEP, ASLR, and a strong implementation of CFI, protecting the program against code injection and control-flow hijacking. We assume that the adversary cannot modify the metadata of our runtime monitor. Protecting metadata is an orthogonal engineering problem and can be solved through, e.g., masking (`and`-ing every memory access), segmentation (for x86-32), protecting the memory region [9], or randomizing the location of sensitive data. Our threat model is a realistic scenario for current attacks and defenses.

4 HexVASAN design

HexVASAN monitors calls to variadic functions and checks for type violations. Since the semantics of how arguments should be interpreted by the function are intrinsic in the logic of the function itself, it is, in general, impossible to determine the number and type of arguments a certain variadic function accepts. For this reason, HexVASAN instruments the code generated by the compiler so that a check is performed at runtime. This check ensures that the arguments consumed by the variadic function match those passed by the caller.

The high level idea is the following: HexVASAN records metadata about the supplied argument types at the call site and verifies that the extracted arguments match in the callee. The number of arguments and their types is always known at the call site and can be encoded efficiently. In the callee this information can then be used to verify individual arguments when they are accessed. To implement such a sanitizer, we must design a metadata store, a pass that instruments call sites, a pass that instruments callers, and a runtime library that manages the metadata store and performs the run-time type verification. Our runtime library aborts the program whenever a mismatch is detected and generates detailed information about the call site and the mismatched arguments.

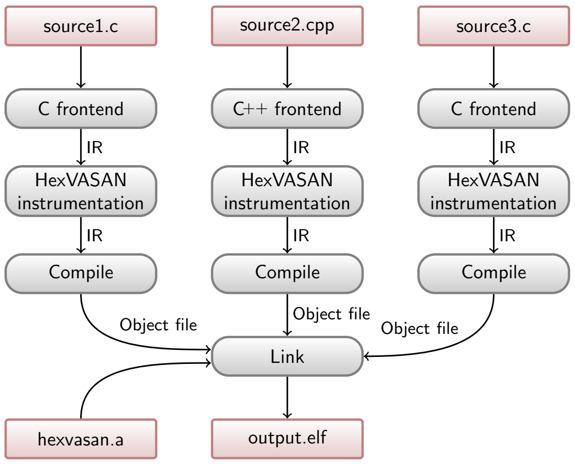

Figure 1: Overview of the HexVASAN compilation pipeline. The HexVASAN instrumentation runs right after the C/C++ frontend, while its runtime library, `hexvasan.a`, is merged into the final executable at link time.

4.1 Analysis and Instrumentation

We designed HexVASAN as a compiler pass to be run in the compilation pipeline right after the C/C++ frontend. The instrumentation collects a set of statically available information about the call sites, encodes it in the LLVM module, and injects calls to our runtime to perform checks during program execution.

Figure 1 provides an overview of the compilation pipeline when HexVASAN is enabled. Source files are first parsed by the C/C++ frontend which generates the intermediate representation on which our instrumentation runs. The normal compilation then proceeds, generating instrumented object files. These object files, along with the HexVASAN runtime library, are then passed to the linker, which creates the instrumented program binary.

4.2 Runtime support

The HexVASAN runtime augments every `va_list` in the original program with the type information generated by our instrumentation pass, and uses this type information to perform run-time type checking on any variadic argument accessed through `va_arg`. By managing the type information in a metadata store, and by maintaining a mapping between `va_lists` and their associated type information, HexVASAN remains fully compatible with the platform ABI. This design also supports interfacing between instrumented programs and non-instrumented libraries.

The HexVASAN runtime manages the type information in two data structures. The core data structure, called the *variadic list map* (VLM), associates `va_list` structures with the type information produced by our instrumentation, and with a counter to track the index of the last argument that was read from the list. A second data structure, the *variadic call stack* (VCS), allows callers of variadic functions to store type information of variadic arguments until the callee initializes the `va_list`.

Each variadic call site is instrumented with a call to `pre_call`, that prepares the information about the call site (a *variadic call site descriptor* or VCSD), and a call to `post_call`, that cleans it up. For each variadic function, the `va_start` calls are instrumented with `list_init`, while `va_copy`, whose purpose is to clone a `va_list`, is instrumented through `list_copy`. The two run-time functions will allocate the necessary data structures to validate individual arguments. Calls to `va_end` are instrumented through `list_end` to free up the corresponding data structures.

Algorithm 1 summarizes the two phases of our analysis and instrumentation pass. The first phase identifies all the calls to variadic functions (both direct and indirect). Note that identifying indirect calls to variadic functions is straight-forward in a compiler framework since, even if the target function is not statically known, its type is. Then, all the parameters passed by that specific call

```
input: a module m
/* Phase 1                                        */
foreach function f in module m do
    foreach variadic call c with n arguments in f do
        vcsd.count ← n;
        foreach argument a of type t do
            vcsd.args.push(t);
        end
        emit call to pre_call(vcsd) before c;
        emit call to post_call() after c;
    end
end
/* Phase 2                                        */
foreach function f in module m do
    foreach call c to va_start(list) do
        emit call to list_init(&list) after c;
    end
    foreach call c to va_copy(dst, src) do
        emit call to list_copy(&dst, &src) after c;
    end
    foreach call c to va_end(list) do
        emit call to list_free(&list) after c;
    end
    foreach call c to va_arg(list, type) do
        emit call to check_arg(&list, type) before c;
    end
end
```

Algorithm 1: The instrumentation process.

site are inspected and recorded, along with their type in a dedicated VCSD which is stored in read-only global data. At this point, a call to `pre_call` is injected before the variadic function call (with the newly created VCSD as a parameter) and, symmetrically, a call to `post_call` is inserted after the call site.

The second phase identifies all calls to `va_start` and `va_copy`, and consequently, the `va_list` variables in the program. Uses of each `va_list` variable are inspected in an architecture-specific way. Once all uses are identified, we inject a call to `check_arg` before dereferencing the argument (which always resides in memory).

4.3 Challenges and Discussion

When designing a variadic function call sanitizer, several issues have to be considered. We highlight details about the key challenges we encountered.

Multiple `va_lists`. Functions are allowed to create multiple `va_lists` to access the same variadic arguments, either through `va_start` or `va_copy` operations. HexVASAN handles this by storing a VLM entry for each individual `va_list`.

Passing `va_lists` as function arguments. While uncommon, variadic functions are allowed to pass the `va_lists` they create as arguments to non-variadic functions. This allows non-variadic functions to access variadic arguments of functions higher in the call stack. Our design takes this into account by maintaining a list map (VLM) and by instrumenting all `va_arg` operations, regardless of whether or not they are in a variadic function.

Multi-threading support. Multiple threads are supported by storing our per-thread runtime state in a thread-local variable as supported on major operating systems.

Metadata format. We use a constant data structure per variadic call site, the VCSD, to hold the number of arguments and a pointer to an array of integers identifying their type. The `check_arg` function therefore only performs two memory accesses, the first to load the number of arguments and the second for the type of the argument currently being checked.

To uniquely identify the data types with an integer, we decided to build a hashing function (described in Algorithm 2) using a set of fixed identifiers for primitive data types and hashing them in different ways depending on how they are aggregated (pointers, `union`, or `struct`). The last hash acts as a terminator marker for aggregate types, which allows us to, e.g., distinguish between {`struct{ int }, int`} and {`struct {struct{ int, int }}`}. Note that an (unlikely) hash collision only results in two different types being accepted as equal. Such a hashing mechanism has the advantage of being deterministic across compilation units, removing the need for

```
input  : a type $t$ and an initial hash value $h$
output: the final hash value $h$
$h = \text{hash}(h, \text{typeID}(t))$;
switch typeID($t$) do
    case AggregateType do
        /* union, struct and pointer */
        foreach $c$ in componentTypes($t$) do
            $h = \text{hashType}(c, h)$;
        end
    case FunctionType do
        $h = \text{hashType}(\text{returnType}(t), h)$;
        foreach $a$ in argTypes($t$) do
            $h = \text{hashType}(a, h)$;
        end
    end
end
$h = \text{hash}(h, \text{typeID}(t))$;
return h
```

Algorithm 2: Algorithm describing the type hashing function *hashType*. *typeID* returns an unique identifier for each basic type (e.g., 32-bit integer, `double`), type of aggregate type (e.g., `struct`, `union`...) and functions. *hash* is a simple hashing function combining two integers. *componentTypes* returns the components of an aggregate type, *returnType* the return type of a function prototype and *argTypes* the type of its arguments.

keeping a global map of type-unique id pairs. Due to the information loss during the translation from C/C++ to LLVM IR, our type system does not distinguish between signed and unsigned types. The required metadata is static and immutable and we mark it as read-only, protecting it from modification. However, the VCS still needs to be protected through other mechanisms.

Handling floating point arguments. In x86-64 ABI, floating point and non-floating point arguments are handled differently. In case of floating point arguments, the first eight arguments are passed in the floating point registers whereas in case of non-floating point the first six are passed in general-purpose registers. HexVASAN handles both argument types.

Support for aggregate data types. According to AMD64 System V ABI, the caller unpacks the fields of the aggregate data types (structs and unions) if the arguments fit into registers. This makes it hard to distinguish between composite types and regular types – if unpacked they are indistinguishable on the callee side from arguments of these types. HexVASAN supports aggregate data types even if the caller unpacks them.

Attacks preserving number and type of arguments. Our mechanism prevents attacks that change the number of arguments or the types of individual arguments.

Format string attacks that only change one modifier can therefore be detected through the type mismatch even if the total number of arguments remains unchanged.

Non-variadic calls to variadic functions. Consider the following code snippet:

```
typedef void (*non_variadic)(int, int);

void variadic(int, ...) { /* ... */ }

int main() {
  non_variadic function_ptr = variadic;
  function_ptr(1, 2);
}
```

In this case, the function call in `main` to `function_ptr` appears to the compiler as a non-variadic function call, since the type of the function pointer is not variadic. Therefore, our pass will not instrument the call site, leading to potential errors.

To handle such (rare) situations appropriately, we would have to instrument all non-variadic call sites too, leading to an unjustified overhead. Moreover, the code above represents *undefined behavior* in C [27, 6.3.2.3p8] and C++ [26, 5.2.10p6], and might not work on certain architectures where the calling convention for variadic and non-variadic function calls are not compatible. The GNU C compiler emits a warning when a function pointer is cast to a different type, therefore we require the developer to correct the code before applying HexVASAN.

Central management of the global state. To allow the HexVASAN runtime to be linked into the base system libraries, such as the C standard library, we made it a static library. Turning the runtime into a shared library is possible, but would prohibit its use during the early process initialization – until the dynamic linker has processed all of the necessary relocations. Our runtime therefore either needs to be added solely to the C standard library (so that it is initialized early in the startup process) or the runtime library must carefully use weak symbols to ensure that each symbol is only defined once if multiple libraries are compiled with our countermeasure.

C++ exceptions and `longjmp`. If an exception is raised while executing a variadic function (or one of its callees), the variadic function may not get a chance to clean up the metadata for any `va_lists` it has initialized, nor may the caller of this variadic function get the chance to clean up the type information it has pushed onto the VCS. Other functions manipulating the thread's stack directly, such as `longjmp`, present similar issues.

C++ exceptions can be handled by modifying the LLVM C++ frontend (i.e., `clang`) to inject an object with a lifetime spanning from immediately before a variadic function call to immediately after. Such an object would call `pre_call` in its constructor and `post_call` in the destructor, leveraging the exception handling mechanism to make HexVASAN exception-safe. Functions like `longjmp` can be instrumented to purge the portions of HexVASAN's data structures that correspond to the discarded stack area. We did not observe any such calls in practice and leave the implementation of handling exceptions and `longjump` across variadic functions as future engineering work.

5 Implementation

We implemented HexVASAN as a sanitizer for the LLVM compiler framework [31], version 3.9.1. We have chosen LLVM for its robust features on analyzing and transforming arbitrary programs as well as extracting reliable type information. The sanitizer can be enabled from the C/C++ frontend (`clang`) by providing the `-fsanitize=vasan` parameter at compile-time. No annotations or other source code changes are required for HexVASAN. Our sanitizer does not require visibility of whole source code (see Section 4.3), but works on individual compilation units. Therefore link-time optimization (LTO) is not required and thus fits readily into existing build systems. In addition, HexVASAN also supports signal handlers.

HexVASAN consists of two components: a static instrumentation pass and a runtime library. The static instrumentation pass works on LLVM IR, adding the necessary instrumentation code to all variadic functions and their callees. The support library is statically linked to the program and, at run-time, checks the number and type of variadic arguments as they are used by the program. In the following we describe the two components in detail.

Static instrumentation. The implementation of the static instrumentation pass follows the description in Section 4. We first iterate through all functions, looking for `CallInst` instructions targeting a variadic function (either directly or indirectly), then we inspect them and create for each one of them a read-only `GlobalVariable` of type `vcsd_t`. As shown in Listing 2, `vcsd_t` is composed by an unsigned integer representing the number of arguments of the considered call site and a pointer to an array (another `GlobalVariable`) with an integer element for each argument of `type_t`. `type_t` is an integer uniquely identifying a data type obtained using the *hashType* function presented in Algorithm 2. At this point a call to `pre_call` is injected before the call site, with the newly create VCSD as a parameter, and a call to `post_call` is injected after the call site.

During the second phase, we first identify all `va_start`, `va_copy`, and `va_end` operations in the program. In the IR code, these operations appear as calls to the LLVM in-

```
struct vcsd_t {
  unsigned count;
  type_t *args;
};

thread_local stack<vcsd_t *> vcs;
thread_local map<va_list *,
                   pair<vcsd_t *, unsigned>> vlm;

void pre_call(vcsd_t *arguments) {
  vcs.push_back(arguments);
}
void post_call() {
  vcs.pop_back();
}
void list_init(va_list *list_ptr) {
  vlm[list_ptr] = { vcs.top(), 0 };
}

void list_free(va_list *list_ptr) {
  vlm.erase(list_ptr);
}

void check_arg(va_list *list_ptr, type_t type) {
  pair<vcsd_t *, unsigned> &args = vlm[list_ptr];
  unsigned index = args.second++;
  assert(index < args.first->count);
  assert(args.first->args[index] == type);
}

int add(int start, ...) {
  /* ... */
  va_start(list, start);
  list_init(&list);
  do {
    check_arg(&list, typeid(int));
    total += va_arg(list, int);
  } while (next != 0);
  va_end(list);
  list_free(&list);
  /* ... */
}

const vcsd_t main_add_vcsd = {
  .count = 3,
  .args = {typeid(int), typeid(int), typeid(int)}
};

int main(int argc, const char *argv[]) {
  /* ... */
  pre_call(&main_add_vcsd);
  int result = add(5, 1, 2, 0);
  post_call();
  printf("%d\n", result);
  /* ... */
}
```

Listing 2: Simplified C++ representation of the instrumented code for Listing 1.

trinsics `llvm.va_start`, `llvm.va_copy`, and `va_end`. We instrument the operations with calls to our runtime's `list_init`, `list_copy`, and `list_free` functions respectively. We then proceed to identify `va_arg` operations. Although the LLVM IR has a dedicated `va_arg` instruction, it is not used on any of the platforms we tested. The `va_list` is instead accessed directly. Our identification of `va_arg` is therefore platform-specific. On x86-64, our primary target, we identify `va_arg` by recognizing accesses to the `gp_offset` and `fp_offset` fields in the x86-64 version of the `va_list` structure (see Section 2.2). The `fp_offset` field is accessed whenever the program attempts to retrieve a floating point argument from the list. The `gp_offset` field is accessed to retrieve any other types of variadic arguments. We insert a call to our runtime's `check_arg` function before the instruction that accesses this field.

Listing 2 shows (in simplified C) how the code in Listing 1 would be instrumented by our sanitizer.

Dynamic variadic type checking. The entire runtime is implemented in plain C code, as this allows it to be linked into the standard C library without introducing a dependency to the standard C++ library. The VCS is implemented as a thread-local stack, and the VLM as a thread-local hash map. The `pre_call` and `post_call` functions push and pop type information onto and from the VCS. The `list_init` function inserts a new entry into the VLM, using the top element on the stack as the entry's type information and initializing the counter for consumed arguments to 0.

`check_arg` looks up the type information for the `va_list` being accessed in the VLM and checks if the requested argument exists (based on the counter of consumed arguments), and if its type matches the one provided by the caller. If either of these checks fails, execution is aborted, and the runtime will generate an error message such as the one shown in Listing 3. As a consequence, the pointer to the argument is never read or written, since the pointer to it is never dereferenced.

```
Error: Type Mismatch
Index is 1
Callee Type : 43 (32-bit Integer)
Caller Type : 15 (Pointer)
Backtrace:
[0] 0x4019ff <__vasan_backtrace+0x1f> at test
[1] 0x401837 <__vasan_check_arg+0x187> at test
[2] 0x8011b3afa <__vfprintf+0x20fa> at libc.so.7
[3] 0x8011b1816 <vfprintf_l+0x86> at libc.so.7
[4] 0x801200e50 <printf+0xc0> at libc.so.7
[5] 0x4024ae <main+0x3e> at test
[6] 0x4012ff <_start+0x17f> at test
```

Listing 3: Error message reported by HexVASAN

6 Evaluation

In this section we present a case study on variadic function based attacks against state-of-the-art CFI implementations. Next, we evaluate the effectiveness of HexVASAN as an exploit mitigation technique. Then, we evaluate the overhead introduced by our HexVASAN prototype implementation on the SPEC CPU2006 integer (CINT2006) benchmarks, on Firefox using standard JavaScript benchmarks, and on micro-benchmarks. We also evaluate how widespread the usage of variadic functions is in SPEC CPU2006 and in Firefox 51.0.1, Chromium 58.0.3007.0, Apache 2.4.23, CPython 3.7.0, nginx 1.11.5, OpenSSL 1.1.1, Wireshark 2.2.1, and the FreeBSD 11.0 base system.

Note that, along with testing the aforementioned software, we also developed an internal set of regression tests. Our regression tests allow us to verify that our sanitizer correctly catches problematic variadic function calls, and does not raise false alarms for benign calls. The test suite explores corner cases, including trying to access arguments that have not been passed and trying to access them using a type different from the one used at the call site.

6.1 Case study: CFI effectiveness

One of the attack scenarios we envision is that an attacker controls the target of an indirect call site. If the intended target of the call site was a variadic function, the attacker could illegally call a different variadic function that expects different variadic arguments than the intended target (yet shares the types for all non-variadic arguments). If the intended target of the call site was a non-variadic function, the attacker could call a variadic function that interprets some of the intended target's arguments as variadic arguments.

All existing CFI mechanisms allow such attacks to some extent. The most precise CFI mechanisms, which rely on function prototypes to classify target sets (e.g., LLVM-CFI, piCFI, or VTV) will allow all targets with the same prototype, possibly restricting to the subset of functions whose addresses are taken in the program. This is problematic for variadic functions, as only non-variadic types are known statically. For example, if a function of type `int (*)(int, ...)` is expected to be called from an indirect call site, then precise CFI schemes allow calls to all other variadic functions of that type, even if those other functions expect different types for the variadic arguments.

A second way to attack variadic functions is to overwrite their arguments directly. This happens, for example, in format string attacks, where an attacker can overwrite the format string to cause misinterpretation of the variadic arguments. HexVASAN detects both of these attacks when the callee attempts to retrieve the variadic arguments using the `va_arg` macro described in Section 2.1. Checking and enforcing the correct types for variadic functions is only possible at runtime and any sanitizer must resort to run-time checks to do so. CFI mechanisms must therefore be extended with a HexVASAN-like mechanism to detect violations. To show that our tool can complement CFI, we create test programs containing several variadic functions and one non-variadic function. The definitions of these functions are shown below.

```
int sum_ints(int n, ...);
int avg_longs(int n, ...);
int avg_doubles(int n, ...);
void print_longs(int n, ...);
void print_doubles(int n, ...);
int square(int n);
```

This program contains one indirect call site from which only the `sum_ints` function can be called legally, and one indirect call site from which only the `square` function can be legally called. We also introduce a memory corruption vulnerability which allows us to override the target of both indirect calls.

We constructed the program such that `sum_ints`, `avg_longs`, `print_longs`, and `square` are all address-taken functions. The `avg_doubles` and `print_doubles` functions are not address-taken.

Functions `avg_longs`, `avg_doubles`, `print_longs`, and `print_doubles` all expect different variadic argument types than function `sum_ints`. Functions `sum_ints`, `avg_longs`, `avg_doubles`, and `square` do, however, all have the same non-variadic prototype (`int (*)(int)`).

We compiled six versions of the test program, instrumenting them with, respectively, HexVASAN, LLVM 3.9 Forward-Edge CFI [59], Per-Input CFI [44], CCFI [35], GCC 6.2's VTV [59] and Visual C++ Control Flow Guard [37]. In each version, we first built an attack involving a variadic function, by overriding the indirect call sites with a call to each of the variadic functions described above. We then also tested overwriting the arguments of the `sum_ints` function, without overwriting the indirect call target. Table 1 shows the detection results.

LLVM Forward-Edge CFI allows calls to `avg_longs` and `avg_doubles` from the `sum_ints` indirect call site because these functions have the same static type signature as the intended call target. This implementation of CFI does not allow calls to variadic functions from non-variadic call sites, however.

CCFI only detects calls to `print_doubles`, a function that is not address-taken and has a different non-variadic prototype than `square`, from the `square` call site. It allows all of the other illegal calls.

	Actual target							
Intended target	Prototype	A.T.?	LLVM-CFI	pi-CFI	CCFI	VTV	CFG	HexVASAN
Variadic	Same	Yes	✗	✗	✗	✗	✗	✓
		No	✗	✓	✗	✗	✗	✓
	Different	Yes	✓	✓	✗	✗	✗	✓
		No	✓	✓	✗	✗	✗	✓
Non-variadic	Same	Yes	✓	✓	✗	✗	✗	✓
		No	✓	✓	✗	✗	✗	✓
	Different	Yes	✓	✓	✗	✗	✗	✓
		No	✓	✓	✓	✗	✗	✓
Original	Overwritten Arguments		✗	✗	✗	✗	✗	✓

Table 1: Detection coverage for several types of illegal calls to variadic functions. ✓ indicates detection, ✗ indicates non-detection. "A.T." stands for *address taken*.

GCC VTV, and Visual C++ CFG allow all of the illegal calls, even if the non-variadic type signature does not match that of the intended call target.

pi-CFI allows calls to the `avg_longs` function from the `sum_ints` indirect call site. `avg_longs` is address-taken and it has the same static type signature as the intended call target. pi-CFI does not allow illegal calls to non-address-taken functions or functions with different static type signatures. pi-CFI also does not allow calls to variadic functions from non-variadic call sites.

All implementations of CFI allow direct overwrites of variadic arguments, as long as the original control flow of the program is not violated.

6.2 Exploit Detection

To evaluate the effectiveness of our tool as a real-world exploit detector, we built a HexVASAN-hardened version of sudo 1.8.3. sudo allows authorized users to execute shell commands as another user, often one with a high privilege level on the system. If compromised, sudo can escalate the privileges of non-authorized users, making it a popular target for attackers. Versions 1.8.0 through 1.8.3p1 of sudo contained a format string vulnerability (CVE-2012-0809) that allowed exactly such a compromise. This vulnerability could be exploited by passing a format string as the first argument (`argv[0]`) of the sudo program. One such exploit was shown to bypass ASLR, DEP, and glibc's FORTIFY_SOURCE protection [20]. In addition, we were able to verify that GCC 5.4.0 and clang 3.8.0 fail to catch this exploit, even when annotating the vulnerable function with the `format` function attribute [5] and setting the compiler's format string checking (`-Wformat`) to the highest level.

Although it is sudo itself that calls the format string function (`fprintf`), HexVASAN can only detect the violation on the callee side. We therefore had to build hardened versions of not just the sudo binary itself, but also the C library. We chose to do this on the FreeBSD platform, as its standard C library can be easily built using LLVM, and HexVASAN therefore readily fits into the FreeBSD build process. As expected, HexVASAN does detect any exploit that triggers the vulnerability, producing the error message shown in Listing 4.

```
$ ln -s /usr/bin/sudo %x%x%x%x
$ ./%x%x%x%x -D9 -A
-------------------------
Error: Index greater than Argument Count
Index is 1
Backtrace:
[0] 0x4053bf <__vasan_backtrace+0x1f> at sudo
[1] 0x405094 <__vasan_check_index+0xf4> at sudo
[2] 0x8015dce24 <__vfprintf+0x2174> at libc.so
[3] 0x8015dac52 <vfprintf_l+0x212> at libc.so
[4] 0x8015daab3 <vfprintf_l+0x73> at libc.so
[5] 0x40bdaf <sudo_debug+0xdf> at sudo
[6] 0x40ada3 <main+0x6c3> at sudo
[7] 0x40494f <_start+0x17f> at sudo
```

Listing 4: Exploit detection in sudo.

6.3 Prevalence of variadic functions

To collect variadic function usage in real software, we extended our instrumentation mechanism to collect statistics about variadic functions and their calls. As shown in Table 2, for each program, we collect:

	Call sites			Func.			Ratio	
Program	Tot.	Ind.	%	Tot.	A.T.	Proto	Tot.	A.T.
Firefox	30225	1664	5.5	421	18	241	1.75	0.07
Chromium	83792	1728	2.1	794	44	396	2.01	0.11
FreeBSD	189908	7508	3.9	1368	197	367	3.73	0.53
Apache	7121	0	0	94	29	41	2.29	0.71
CPython	4183	0	0	382	0	38	10.05	0.00
nginx	1085	0	0	26	0	14	1.86	0.00
OpenSSL	4072	1	0.02	23	0	15	1.53	0.00
Wireshark	37717	0	0	469	1	110	4.26	0.01
perlbench	1460	1	0.07	60	2	18	3.33	0.11
bzip2	85	0	0	3	0	3	1.00	0.00
gcc	3615	55	1.5	125	0	31	4.03	0.00
mcf	29	0	0	3	0	3	1.00	0.00
milc	424	0	0	21	0	8	2.63	0.00
namd	485	0	0	24	2	8	3.00	0.25
gobmk	2911	0	0	35	0	8	4.38	0.00
soplex	6	0	0	2	1	2	1.00	0.50
povray	1042	40	3.8	45	10	16	2.81	0.63
hmmer	671	7	1	9	1	5	1.80	0.20
sjeng	253	0	0	4	0	3	1.33	0.00
libquantum	74	0	0	91	0	7	13.00	0.00
h264ref	432	0	0	85	5	13	6.54	0.38
lbm	11	0	0	3	0	2	1.50	0.00
omnetpp	340	0	0	48	23	19	2.53	1.21
astar	42	0	0	4	1	4	1.00	0.25
sphinx3	731	0	0	20	0	5	4.00	0.00
xalancbmk	19	0	0	4	2	4	1.00	0.50

Table 2: Statistics of Variadic Functions for Different Benchmarks. The second and third columns are variadic call sites broken into "Tot." (total) and "Ind." (indirect); % shows the percentage of variadic call sites. The fifth and sixth columns are for variadic functions. "A.T." stands for *address taken*. "Proto." is the number of distinct variadic function prototypes. "Ratio" indicates the *function-per-prototypes* ratio for variadic functions.

Call sites. The number of function calls targeting variadic functions. We report the total number and how many of them are indirect, since they are of particular interest for an attack scenario where the adversary can override a function pointer.

Variadic functions. The number of variadic functions. We report their total number and how many of them have their address taken, since CFI mechanism cannot prevent functions with their address taken from being reachable from indirect call sites.

Variadic prototypes. The number of distinct variadic function prototypes in the program.

Functions-per-prototype. The average number of variadic functions sharing the same prototype. This measures how many targets are available, on average, for each indirect call sites targeting a specific prototype. In practice, this the average number of permitted destinations for an indirect call site in the case of a perfect CFI implementation. We report this value both considering all the variadic functions and only those whose address is taken.

Interestingly, each benchmark we analyzed contains calls to variadic functions and several programs (Firefox, OpenSSL, perlbench, gcc, povray, and hmmer) even contain indirect calls to variadic functions. In addition to *calling* variadic functions, each benchmark also *defines* numerous variadic functions (421 for Firefox, 794 for Chromium, 1368 for FreeBSD, 469 for Wireshark, and 382 for CPython). Variadic functions are therefore prevalent and used ubiquitously in software. Adversaries have plenty of opportunities to modify these calls and to attack the implicit contract between caller and callee. The compiler is unable to enforce any static safety guarantees when calling these functions, either for the number of arguments, nor their types. In addition, many of the benchmarks have variadic functions that are called indirectly, often with their address being taken. Looking at Firefox, a large piece of software, the numbers are even more staggering with several thousand indirect call sites that target variadic functions and 241 different variadic prototypes.

The prevalence of variadic functions leaves both a large attack surface for attackers to either redirect variadic calls to alternate locations (even if defense mechanisms like CFI are present) or to modify the arguments so that callees misinterpret the supplied arguments (similar to extended format string attacks).

In addition, the compiler has no insight into these functions and cannot statically check if the programmer supplied the correct parameters. Our sanitizer identified three interesting cases in `omnetpp`, one of the SPEC CPU2006 benchmarks that implements a discrete event simulator. The benchmark calls a variadic functions with a mismatched type, where it expects a `char *` but receives a NULL, which has type `void *`. Listing 5 shows the offending code.

We also identified a bug in SPEC CPU2006's `perlbench`. This benchmark passes the result of a subtraction of two character pointers as an argument to a

```
static sEnumBuilder _EtherMessageKind(
    "EtherMessageKind",
   JAM_SIGNAL, "JAM_SIGNAL",
   ETH_FRAME, "ETH_FRAME",
   ETH_PAUSE, "ETH_PAUSE",
   ETHCTRL_DATA, "ETHCTRL_DATA",
   ETHCTRL_REGISTER_DSAP,
       "ETHCTRL_REGISTER_DSAP",
   ETHCTRL_DEREGISTER_DSAP,
       "ETHCTRL_DEREGISTER_DSAP",
   ETHCTRL_SENDPAUSE, "ETHCTRL_SENDPAUSE",
   0, NULL
);
```

Listing 5: Variadic violation in omnetpp.

variadic function. At the call site, this argument is a machine word-sized integer (i.e., 64-bits integer on our test platform). The callee truncates this argument to a 32-bit integer by calling `va_arg(list, int)`. HexVASAN reports this (likely unintended) truncation as a violation.

6.4 Firefox

We evaluate the performance of HexVASAN by instrumenting Firefox (51.0.1) and using three different browser benchmark suites: Octane, JetStream, and Kraken. Table 3 shows the comparison between the HexVASAN instrumented Firefox and native Firefox. To reduce variance between individual runs, we averaged fifteen runs for each benchmark (after one warmup run). For each run we started Firefox, ran the benchmark, and closed the browser. HexVASAN incurs only 1.08% and 1.01% overhead for Octane and JetStream respectively and speeds up around 0.01% for Kraken. These numbers are indistinguishable from measurement noise. Octane [4] and JetStream measure the time a test takes to complete and then assign a score that is inversely proportional to the runtime, whereas Kraken [3] measures the speed of test cases gathered from different real-world applications and libraries.

Benchmark		Native	HexVASAN
Octane	AVERAGE	31241.80	30907.73
	STDDEV	2449.82	2442.82
	OVERHEAD		-1.08%
JetStream	AVERAGE	200.76	198.75
	STDDEV	0.66	1.68
	OVERHEAD		-1.01%
Kraken	AVERAGE [ms]	832.48	832.41
	STDDEV [ms]	7.41	12.71
	OVERHEAD		0.01%

Table 3: Performance overhead on Firefox benchmarks. For Octane and JetStream higher is better, while for Kraken lower is better.

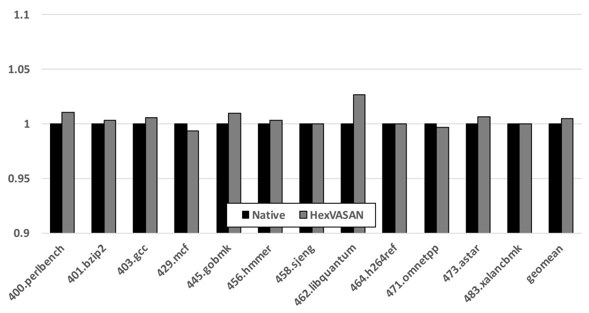

Figure 2: Run-time overhead of HexVASAN in the SPECint CPU2006 benchmarks, compared to baseline LLVM 3.9.1 performance.

6.5 SPEC CPU2006

We measured HexVASAN's run-time overhead by running the SPEC CPU2006 integer (CINT2006) benchmarks on an Ubuntu 14.04.5 LTS machine with an Intel Xeon E5-2660 CPU and 64 GiB of RAM. We ran each benchmark program on its reference inputs and measured the average run-time over three runs. Figure 2 shows the results of these tests. We compiled each benchmark with a vanilla clang/LLVM 3.9.1 compiler and optimization level `-O3` to establish a baseline. We then compiled the benchmarks with our modified clang/LLVM 3.9.1 compiler to generate the HexVASAN results.

The geometric mean overhead in these benchmarks was just 0.45%, indistinguishable from measurement noise. The only individual benchmark result that stands out is that of `libquantum`. This benchmark program performed 880M variadic function calls in a run of just 433 seconds.

6.6 Micro-benchmarks

Besides evaluating large benchmarks, we have also measured HexVASAN's runtime overhead on a set of micro-benchmarks. We have written test cases for variadic functions with different number of arguments, in which we repeatedly invoke the variadic functions. Table 4 shows the comparison between the native and HexVASAN-instrumented micro-benchmarks. Overall, HexVASAN incurs runtime overheads of 4-6x for variadic function calls due to the additional security checks. In real-world programs, however, variadic functions are invoked rarely, so HexVASAN has little impact on the overall runtime performance.

	# calls	Native [μs]	HexVASAN [μs]
Variadic function argument count: 3	1	0	0
	100	2	12
	1000	20	125
Variadic function argument count: 12	1	0	0
	100	6	22
	1000	55	198

Table 4: Performance overhead in micro-benchmarks.

7 Related work

HexVASAN can either be used as an always-on runtime monitor to mitigate exploits or as a sanitizer to detect bugs, sharing similarities with the sanitizers that exist primarily in the LLVM compiler. Similar to HexVASAN, these sanitizers embed run-time checks into a program by instrumenting potentially dangerous program instructions.

AddressSanitizer [54] (ASan), instruments memory accesses and allocation sites to detect spatial memory errors, such as out-of-bounds accesses, as well as temporal memory errors, such as use-after-free bugs. Undefined Behavior Sanitizer [52] (UBSan) instruments various types of instructions to detect operations whose semantics are not strictly defined by the C and C++ standards, e.g., increments that cause signed integers to overflow, or null-pointer dereferences. Thread Sanitizer [55] (TSAN) instruments memory accesses and atomic operations to detect data races, deadlocks, and various misuses of synchronization primitives. Memory Sanitizer [58] (MSAN) detects uses of uninitialized memory.

CaVer [32] is a sanitizer targeted at verifying correctness of downcasts in C++. Downcasting converts a base class pointer to a derived class pointer. This operation may be unsafe as it cannot be statically determined, in general, if the pointed-to object is of the derived class type. TypeSan [25] is a refinement of CaVer that reduces overhead and improves the sanitizer coverage.

UniSan [34] sanitizes information leaks from the kernel. It ensures that data is initialized before leaving the kernel, preventing reads of uninitialized memory.

All of these sanitizers are highly effective at finding specific types of bugs, but, unlike HexVASAN, they do not address misuses of variadic functions. The aforementioned sanitizers also differ from HexVASAN in that they typically incur significant run-time and memory overhead.

Different control-flow hijacking mitigations offer partial protection against variadic function attacks by preventing adversaries from calling variadic functions through control-flow edges that do not appear in legitimate executions of the program. Among these mitigations, we find Code Pointer Integrity (CPI) [30], a mitigation that prevents attackers from overwriting code pointers in the program, and various implementations of Control-Flow Integrity (CFI), a technique that does not prevent code pointer overwrites, but rather verifies the integrity of control-flow transfers in the program [6, 7, 11, 14–16, 21, 22, 28, 35, 37, 38, 41–44, 46, 49–51, 59, 61–66].

Control-flow hijacking mitigations *cannot* prevent attackers from overwriting variadic arguments directly. At best, they can prevent variadic functions from being called through control-flow edges that do not appear in legitimate executions of the program. We therefore argue that HexVASAN and these mitigations are orthogonal. Moreover, prior research has shown that many of the aforementioned implementations fail to fully prevent control-flow hijacking as they are too imprecise [8, 17, 19, 23], too limited in scope [53, 57], vulnerable to information leakage attacks [18], or vulnerable to spraying attacks [24, 45]. We further showed in Section 6.1 that variadic functions exacerbate CFI's imprecision problems, allowing additional leeway for adversaries to attack variadic functions.

Defenses that protect against direct overwrites or misuse of variadic arguments have thus far only focused on format string attacks, which are a subset of the possible attacks on variadic functions. LibSafe detects potentially dangerous calls to known format string functions such as `printf` and `sprintf` [60]. A call is considered dangerous if a `%n` specifier is used to overwrite the frame pointer or return address, or if the argument list for the `printf` function is not contained within a single stack frame. FormatGuard [12] instruments calls to `printf` and checks if the number of arguments passed to `printf` matches the number of format specifiers used in the format string.

Shankar et al. proposed to use static taint analysis to detect calls to format string functions where the format string originates from an untrustworthy source [56]. This approach was later refined by Chen and Wagner [10] and used to analyze thousands of packages in the Debian 3.1 Linux distribution. TaintCheck [39] also detects untrustworthy format strings, but relies on dynamic taint analysis to do so.

`_FORTIFY_SOURCE` of *glibc* provides some lightweight checks to ensure all the arguments are consumed. However, it can be bypassed [2] and does not check for type-mismatch. Hence, none of these aforementioned solutions provide comprehensive protection against variadic argument overwrites or misuse.

8 Conclusions

Variadic functions introduce an implicitly defined contract between the caller and callee. When the programmer fails to enforce this contract correctly, the violation leads to runtime crashes or opens up a vulnerability to an attacker. Current tools, including static type checkers and CFI implementations, do not find variadic function type errors or prevent attackers from exploiting calls to variadic functions. Unfortunately, variadic functions are prevalent. Programs such as SPEC CPU2006, Firefox, Apache, CPython, nginx, wireshark and libraries frequently leverage variadic functions to offer flexibility and abundantly call these functions.

We have designed a sanitizer, HexVASAN, that addresses this attack vector. HexVASAN is a light weight runtime monitor that detects bugs in variadic functions and prevents the bugs from being exploited. It imposes negligible overhead (0.45%) on the SPEC CPU2006 benchmarks and is effective at detecting type violations when calling variadic arguments. Download HexVASAN at `https://github.com/HexHive/HexVASAN`.

9 Acknowledgments

We thank the anonymous reviewers for their insightful comments. We also thank our shepherd Adam Doupé for his informative feedback. This material is based in part upon work supported by the National Science Foundation under awards CNS-1513783, CNS-1657711, and CNS-1619211, by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-15-C-0124 and FA8750-15-C-0085, and by Intel Corporation. We also gratefully acknowledge a gift from Oracle Corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Defense Advanced Research Projects Agency (DARPA) and its Contracting Agents, or any other agency of the U.S. Government.

References

[1] `http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8617`.

[2] A eulogy for format strings. `http://phrack.org/issues/67/9.html`.

[3] Kraken benchmark. `https://wiki.mozilla.org/Kraken`.

[4] Octane benchmark. `https://developers.google.com/octane/faq`.

[5] Using the gnu compiler collection (gcc) - function attributes. `https://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Function-Attributes.html`.

[6] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)* (2005).

[7] Bounov, D., Kici, R., and Lerner, S. Protecting C++ dynamic dispatch through vtable interleaving. In *Symposium on Network and Distributed System Security (NDSS)* (2016).

[8] Carlini, N., Barresi, A., Payer, M., Wagner, D., and Gross, T. R. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security Symposium* (2015).

[9] Castro, M., Costa, M., Martin, J.-P., Peinado, M., Akritidis, P., Donnelly, A., Barham, P., and Black, R. Fast byte-granularity software fault isolation. In *ACM Symposium on Operating Systems Principles (SOSP)* (2009).

[10] Chen, K., and Wagner, D. Large-scale analysis of format string vulnerabilities in debian linux. In *Proceedings of the 2007 workshop on Programming languages and analysis for security* (2007).

[11] Cheng, Y., Zhou, Z., Miao, Y., Ding, X., and Deng, R. H. ROPecker: A generic and practical approach for defending against ROP attacks. In *Symposium on Network and Distributed System Security (NDSS)* (2014).

[12] Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G., Frantzen, M., and Lokier, J. Formatguard: Automatic protection from printf format string vulnerabilities. In *USENIX Security Symposium* (2001).

[13] Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., and Hinton, H. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium* (1998).

[14] Criswell, J., Dautenhahn, N., and Adve, V. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *IEEE Symposium on Security and Privacy (S&P)* (2014).

[15] Davi, L., Dmitrienko, A., Egele, M., Fischer, T., Holz, T., Hund, R., Nürnberger, S., and Sadeghi, A.-R. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Symposium on Network and Distributed System Security (NDSS)* (2012).

[16] Davi, L., Koeberl, P., and Sadeghi, A.-R. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Annual Design Automation Conference (DAC)* (2014).

[17] Davi, L., Sadeghi, A.-R., Lehmann, D., and Monrose, F. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium* (2014).

[18] Evans, I., Fingeret, S., González, J., Otgonbaatar, U., Tang, T., Shrobe, H., Sidiroglou-Douskos, S., Rinard, M., and Okhravi, H. Missing the point (er): On the effectiveness of code pointer integrity. In *IEEE Symposium on Security and Privacy (S&P)* (2015).

[19] Evans, I., Long, F., Otgonbaatar, U., Shrobe, H., Rinard, M., Okhravi, H., and Sidiroglou-Douskos, S. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *ACM Conference on Computer and Communications Security (CCS)* (2015).

[20] Exploit Database. sudo_debug privilege escalation. `https://www.exploit-db.com/exploits/25134/`, 2013.

[21] Gawlik, R., and Holz, T. Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs. In *Annual Computer Security Applications Conference (ACSAC)* (2014).

[22] Ge, X., Talele, N., Payer, M., and Jaeger, T. Fine-Grained Control-Flow Integrity for Kernel Software. In *IEEE European Symp. on Security and Privacy* (2016).

[23] GÖKTAS, E., ATHANASOPOULOS, E., BOS, H., AND PORTOKALIDIS, G. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy (S&P)* (2014).

[24] GÖKTAS, E., GAWLIK, R., KOLLENDA, B., ATHANASOPOULOS, E., PORTOKALIDIS, G., GIUFFRIDA, C., AND BOS, H. Undermining information hiding (and what to do about it). In *USENIX Security Symposium* (2016).

[25] HALLER, I., JEON, Y., PENG, H., PAYER, M., GIUFFRIDA, C., BOS, H., AND VAN DER KOUWE, E. Typesan: Practical type confusion detection. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[26] Information technology – Programming languages – C++. Standard, International Organization for Standardization, Geneva, CH, Dec. 2014.

[27] Information technology – Programming languages – C. Standard, International Organization for Standardization, Geneva, CH, Dec. 2011.

[28] JANG, D., TATLOCK, Z., AND LERNER, S. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks. In *Symposium on Network and Distributed System Security (NDSS)* (2014).

[29] JELINEK, J. FORTIFY_SOURCE. `https://gcc.gnu.org/ml/gcc-patches/2004-09/msg02055.html`, 2004.

[30] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code-pointer integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).

[31] LATTNER, C., AND ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2004).

[32] LEE, B., SONG, C., KIM, T., AND LEE, W. Type casting verification: Stopping an emerging attack vector. In *USENIX Security Symposium* (2015).

[33] LINUX PROGRAMMER'S MANUAL. va_start (3) - Linux Manual Page.

[34] LU, K., SONG, C., KIM, T., AND LEE, W. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[35] MASHTIZADEH, A. J., BITTAU, A., BONEH, D., AND MAZIÈRES, D. Ccfi: cryptographically enforced control flow integrity. In *ACM Conference on Computer and Communications Security (CCS)* (2015).

[36] MATZ, M., HUBICKA, J., JAEGER, A., AND MITCHELL, M. System v application binary interface. *AMD64 Architecture Processor Supplement, Draft v0.99* (2013).

[37] MICROSOFT CORPORATION. Control Flow Guard (Windows). `https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx`, 2016.

[38] MOHAN, V., LARSEN, P., BRUNTHALER, S., HAMLEN, K., AND FRANZ, M. Opaque control-flow integrity. In *Symposium on Network and Distributed System Security (NDSS)* (2015).

[39] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Symposium on Network and Distributed System Security (NDSS)* (2005).

[40] NISSIL, R. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1886`.

[41] NIU, B., AND TAN, G. Monitor integrity protection with space efficiency and separate compilation. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[42] NIU, B., AND TAN, G. Modular control-flow integrity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2014).

[43] NIU, B., AND TAN, G. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)* (2014).

[44] NIU, B., AND TAN, G. Per-input control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)* (2015).

[45] OIKONOMOPOULOS, A., ATHANASOPOULOS, E., BOS, H., AND GIUFFRIDA, C. Poking holes in information hiding. In *USENIX Security Symposium* (2016).

[46] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security Symposium* (2013).

[47] PAX TEAM. Pax address space layout randomization (aslr).

[48] PAX TEAM. PaX non-executable pages design & implementation. `http://pax.grsecurity.net/docs/noexec.txt`, 2004.

[49] PAYER, M., BARRESI, A., AND GROSS, T. R. Fine-grained control-flow integrity through binary hardening. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2015).

[50] PEWNY, J., AND HOLZ, T. Control-flow restrictor: Compiler-based CFI for iOS. In *Annual Computer Security Applications Conference (ACSAC)* (2013).

[51] PRAKASH, A., HU, X., AND YIN, H. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Symposium on Network and Distributed System Security (NDSS)* (2015).

[52] PROJECT, G. C. Undefined behavior sanitizer. `https://www.chromium.org/developers/testing/undefinedbehaviorsanitizer`.

[53] SCHUSTER, F., TENDYCK, T., LIEBCHEN, C., DAVI, L., SADEGHI, A.-R., AND HOLZ, T. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *IEEE Symposium on Security and Privacy (S&P)* (2015).

[54] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., AND VYUKOV, D. Addresssanitizer: a fast address sanity checker. In *USENIX Annual Technical Conference* (2012).

[55] SEREBRYANY, K., AND ISKHODZHANOV, T. Threadsanitizer: Data race detection in practice. In *Workshop on Binary Instrumentation and Applications* (2009).

[56] SHANKAR, U., TALWAR, K., FOSTER, J. S., AND WAGNER, D. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium* (2001).

[57] SNOW, K. Z., MONROSE, F., DAVI, L., DMITRIENKO, A., LIEBCHEN, C., AND SADEGHI, A. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy (S&P)* (2013).

[58] STEPANOV, E., AND SEREBRYANY, K. Memorysanitizer: Fast detector of uninitialized memory use in c++. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2015).

[59] TICE, C., ROEDER, T., COLLINGBOURNE, P., CHECKOWAY, S., ERLINGSSON, Ú., LOZANO, L., AND PIKE, G. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security Symposium* (2014).

[60] TSAI, T., AND SINGH, N. Libsafe 2.0: Detection of format string vulnerability exploits. *white paper, Avaya Labs* (2001).

[61] VAN DER VEEN, V., ANDRIESSE, D., GÖKTAŞ, E., GRAS, B., SAMBUC, L., SLOWINSKA, A., BOS, H., AND GIUFFRIDA, C. PathArmor: Practical ROP protection using context-sensitive CFI. In *ACM Conference on Computer and Communications Security (CCS)* (2015).

[62] WANG, Z., AND JIANG, X. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE Symposium on Security and Privacy (S&P)* (2010).

[63] YUAN, P., ZENG, Q., AND DING, X. Hardware-assisted fine-grained code-reuse attack detection. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)* (2015).

[64] ZHANG, C., SONG, C., CHEN, K. Z., CHEN, Z., AND SONG, D. VTint: Defending virtual function tables' integrity. In *Symposium on Network and Distributed System Security (NDSS)* (2015).

[65] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical control flow integrity and randomization for binary executables. In *IEEE Symposium on Security and Privacy (S&P)* (2013).

[66] ZHANG, M., AND SEKAR, R. Control flow integrity for cots binaries. In *USENIX Security Symposium* (2013).