Venerable Variadic Vulnerabilities Vanquished

*Priyam Biswas**[*] Alessandro Di Federico[*†] Scott A. Carr[*] Prabhu Rajasekaran[§]
Stijn Volckaert[§] Yeoul Na[§] Michael Franz[§] Mathias Payer[*]

[*]Purdue University [†] Politecnico di Milano [§]University of California, Irvine

Variadic Function

- C and C++ support variadic functions
- Variable number of arguments
- Implicit contract between caller and callee
- Cannot statically check the argument types

```c
int add(int n, ...)
{

    va_list list;
    va_start(list, n);
    for (int i=0; i < n; i++)
       total=total + va_arg(list, int);

    va_end(list);
    return total;
}

int main(int argc, const char * argv[])
{

    result = add(3, val1, val2, val3);
    result = add(2, val1, val2);
    return 0;
}
```

Motivation

- Parameters of variadic functions cannot be statically checked

- Attacks violate the implicit contract between caller and callee
  - Attacks cause disparity: more/less arguments or wrong argument type

- Existing defenses do not prevent such attacks

Prevalence of Variadic Functions

Program	Call Sites		Functions		Prototype
	Total	Indirect	Total	Address Taken	
Firefox	30,225	1,664	421	18	241
Chromium	83,792	1,728	794	44	396
FreeBSD	189,908	7,508	1,368	197	367
Apache	7,121	0	94	29	41
CPython	4,183	0	382	0	38
Nginx	1,085	0	26	0	14
OpenSSL	4,072	1	23	0	15
Wireshark	37,717	0	469	1	110

Threat Model

- Program contains arbitrary memory corruption

- Existing defense mechanisms such as DEP, ASLR, CFI are deployed

- Capabilities of the attacker
  - Directly overwriting the arguments of a variadic function
  - Hijacking indirect calls and call variadic functions over control-flow edges

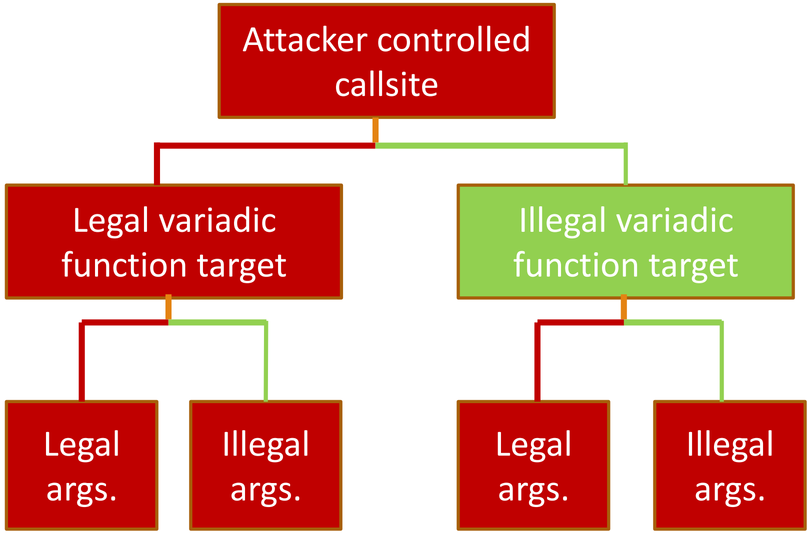
Control Flow Integrity (CFI)

- Verifies indirect control flow transfers based on statically determined set

- Allows all targets with the same prototype

```
int foo (int n, …)
int baz(int n, …)
int bar(int n, …)
int boo (n)
void func(int n, …)
Void func2(int n, …)
```

int*(int)



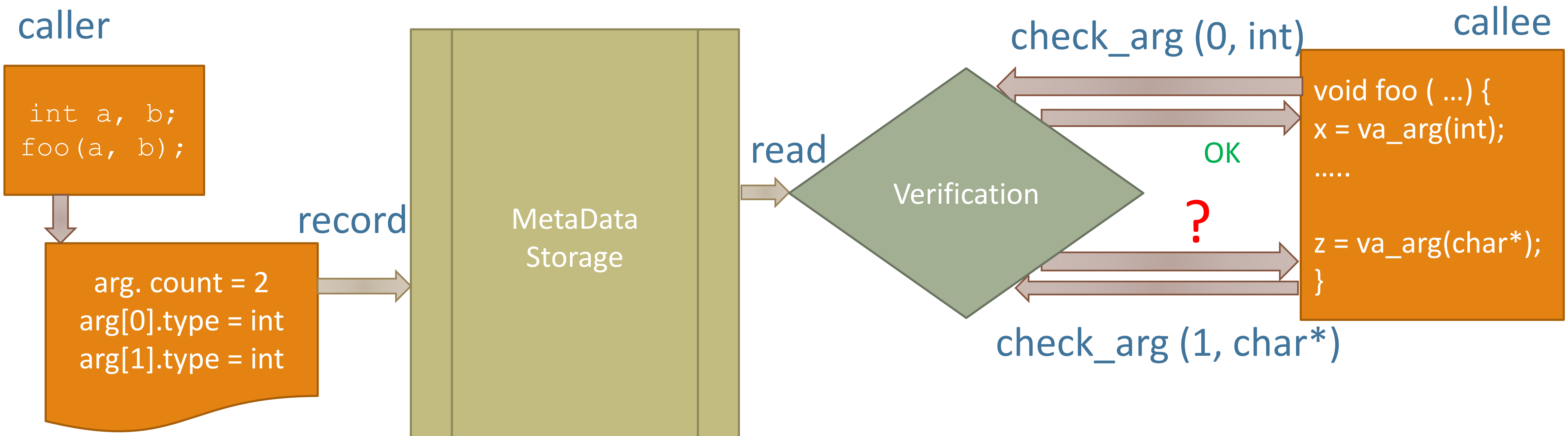
Intended target	Actual target		LLVM-CFI [1]	pi-CFI [2]	CCFI [3]	VTV [4]	CFG [5]	HexVASAN
	Prototype	Addr. Taken						
Variadic	Same	Yes	X	X	X	X	X	√
		No	X	√	X	X	X	√
	Different	Yes	√	√	X	X	X	√
		No	√	√	X	X	X	√
Non-Variadic	Same	Yes	√	√	X	X	X	√
		No	√	√	X	X	X	√
	Different	Yes	√	√	X	X	X	√
		No	√	√	√	X	X	√
Original	Overwritten Arguments		X	X	X	X	X	√

1. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM , USENIX Security 2014
2. Per-Input Control-Flow Integrity, CCS 2015
3. CCFI: Cryptographically Enforced Control Flow Integrity, CCS 2015
4. GCC 6.2 Virtual Table Verification
5. Microsoft Corporation: Control Flow Guard (Windows)

Our Approach

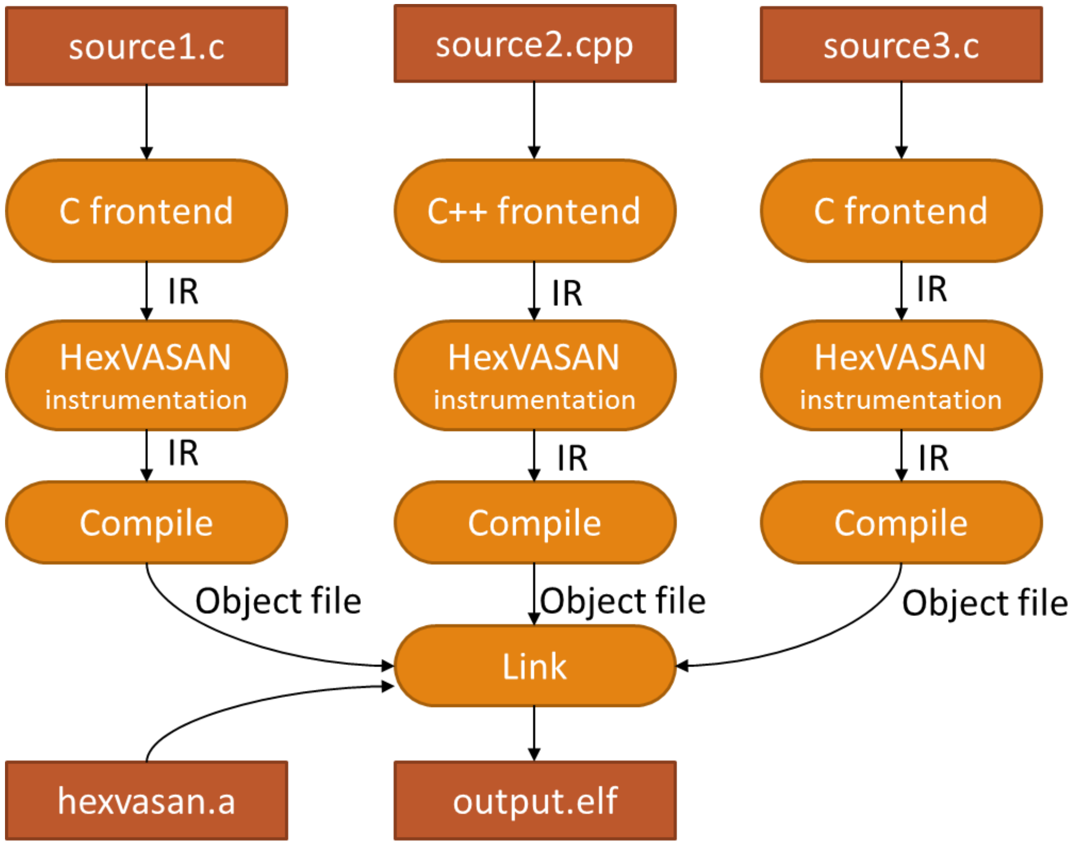
- Enforce contract between caller and callee
- Verify argument types at runtime
- Abort if there is an error

HexVASAN Design



Implementation

- Implemented as LLVM pass
- Statically instrument code
- Dynamically verify types of variadic arguments (library)



Real Code Is Hard!

- Handling multiple va_list
  - HexVASAN supports it by recording each va_list separately
- Floating-point arguments
  - Handles floating point and non-floating point arguments separately
- Handling aggregate data types
  - Caller unpacks the fields if arguments fit into registers
  - Traces back to get the correct data type

Evaluation

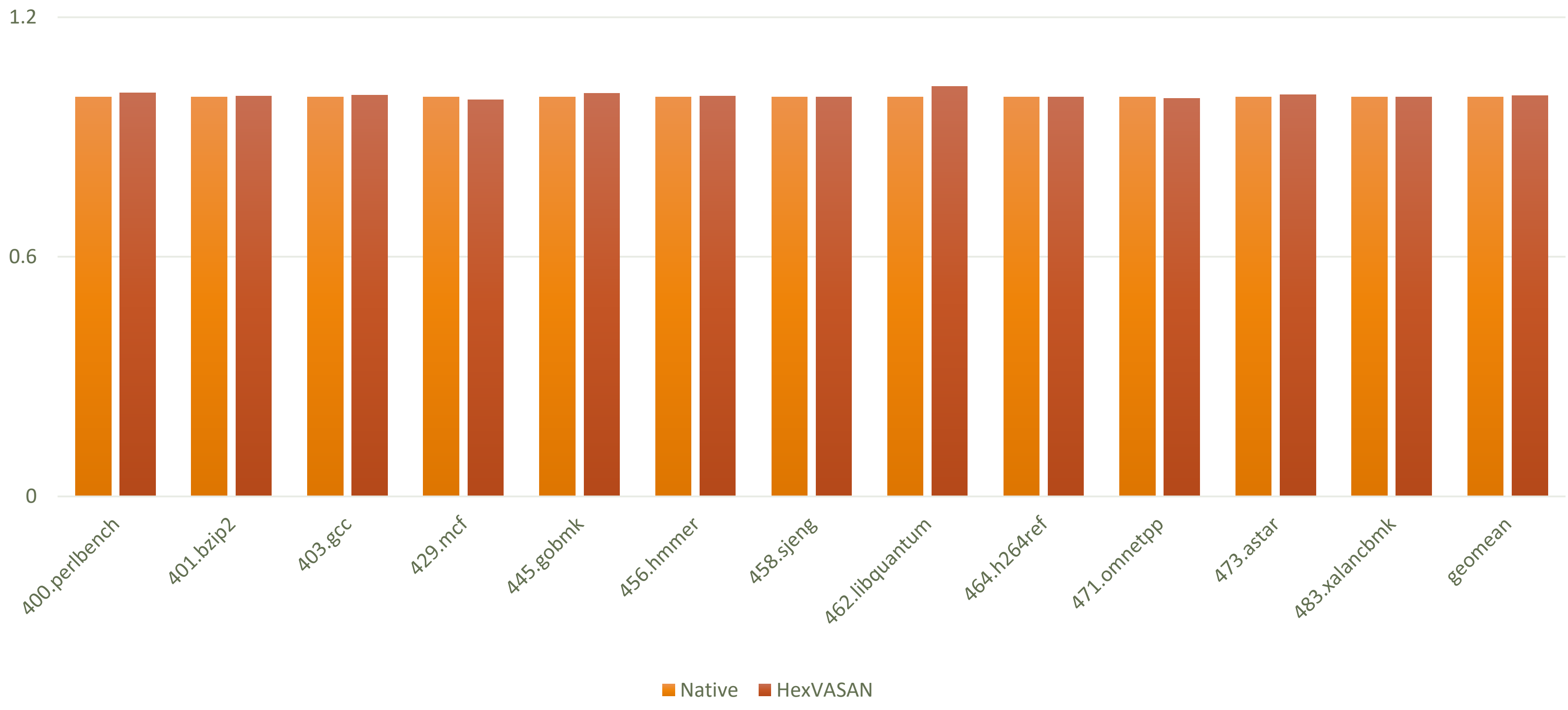
- Comparison with state-of-the-art CFI mechanisms
- Usage of variadic functions in existing software
- Performance overhead in SPEC CPU2006 benchmark & Firefox

Exploit Detection

- Format string vulnerability in “sudo” CVE-2012-0809
- Attacker can escalate the privileges
- Not detected by -Wformat
- HexVASAN detects exploit

```
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
```

Performance Overhead: SPEC CPU2006



Interesting Cases: Spec CPU2006

- Omnetpp
  - Caller : `NULL`
  - Callee : `char*`
- Perlbench
  - Caller : Subtraction of two `char` pointers (64 bit)
  - Callee : `int` (32 bit)

Performance Overhead: Firefox

Benchmark		Native	HexVASAN
Octane	AVERAGE	33,824.40	33717.40
	STDDEV	74.96	125.89
	OVERHEAD		0.32%
JetStream	AVERAGE	194.86	193.68
	STDDEV	1.30	0.58
	OVERHEAD		0.61%
Kraken	AVERAGE	885.52	887.12
	STDDEV	11.02	7.31
	OVERHEAD		0.18%

Sample Findings: Firefox

- Case 1
  - Caller : `unsigned long`
  - Callee : `unsigned int`
- Case 2
  - Caller : `Bool`
  - Callee : `unsigned long`
- Case 3
  - Caller : `void*`
  - Callee : `unsigned long`

Conclusion

- HexVASAN successfully monitors variadic arguments
- Detects bugs due to type mismatch in variadic functions
- Negligible overhead in SPEC CPU2006 and Firefox
- Open Source at https://github.com/HexHive/HexVASAN

Thank you! Questions?

Open Source at https://github.com/HexHive/HexVASAN

```c
int add(int n, ...)
{

    va_list list;
    va_start(list, n);
    for (int i=0; i < n; i++)
       total=total + va_arg(list, int);

    va_end(list);
    return total;
}

int main(int argc, const char * argv[])
{

    result = add(3, val1, val2, val3);
    return 0;
}
```

```c
int add(int n, ...)
{

    va_list list;
    va_start(list, n);
    list_init(&list);
    for (int i=0; i < n; i++) {
       check_arg(&list, typeid(int));
       total=total + va_arg(list, int);}
    va_end(list);
    list_free(&list);
    return total;
}

int main(int argc, const char * argv[])
{
    precall(vcsd);
    result = add(3, val1, val2, val3);
    postcall(vcsd);
    return 0;
}
```