Memory Categorization: Separating Attacker-Controlled Data

Matthias Neugschwandtner, Alessandro Sorniotti, and Anil Kurmus

IBM Research – Zurich

Abstract. Memory corruption attacks against software written in C or C++ are still prevalent and remain a significant cause of security breaches. Defenses providing full memory safety remain expensive, and leaner defenses only addressing control-flow data are insufficient.
We introduce *memory categorization*, an approach to separate data based on attacker control to mitigate the exploitation of memory corruption vulnerabilities such as use-after-free and use-after-return. MEMCAT implements this approach by: *i*) providing separate memory allocators for different data categories, *ii*) categorizing the use of memory allocations, *iii*) changing allocations to take advantage of the categorization.
We demonstrate the effectiveness of MEMCAT in a case study on actual vulnerabilities in real-world programs. We further show that, although our prototype implementation causes a high overhead in two edge cases, in most cases the performance hit remains negligible, with a median overhead of less than 3% on the SPEC benchmark suite.

1 Introduction

Most prominent published exploits (e.g., through competitions or various vulnerability reward programs) in the past few years rely on memory corruption vulnerabilities to achieve remote code execution, sandbox escape, privilege escalation, or leakage of sensitive data. The increasing difficulty of crafting such exploits is in part due to mitigations that were developed in the past two decades. These include advanced defense mechanisms that were pioneered by the research community, such as Control Flow Integrity (CFI) [**cfi:abadi05**].

Many mitigation approaches focus on providing control-flow integrity, i.e., protecting code and code pointers. CFI approaches often assume a very powerful attacker, capable of arbitrary memory reads and writes, albeit with a comparatively restrictive goal: modification of control flow. However, vulnerabilities such as Heartbleed demonstrate that even attackers with (restricted) out-of-bound read capability can already achieve their goals (such as leaking sensitive cryptographic material). In essence, control-flow data is in general not the only data that a program needs to protect to fulfil its security goals [**chen05ncd**]. At the same time, approaches that aim at providing full memory safety [**nagarakatte09softbound**, **nagarakatte10cets**] currently incur prohibitively high overhead.

A number of mitigation-enabling techniques, or selective hardening techniques, have been proposed to achieve a better trade-off between the performance of lightweight

hardening mechanisms and the security of full memory safety. ASAP [**wagner15asap**] statically removes safety checks in "hot" code, while Split Kernel [**kurmus14ccs**], PartiSan [**lettner18raid**], BinRec [**kroes18feast**] remove safety checks dynamically. However, these solutions all apply hardening based on categorizing *code* as either performance sensitive, or security sensitive. We argue that categorizing *data* is a better match, albeit more challenging to achieve.

In this paper, we introduce *memory categorization*, a new mitigation-enabling technique that separates *attacker-controlled* data from other data, including internal program data. Memory categorization is in part motivated by a simple, but powerful observation: an attacker that can only read or modify its own attacker-controlled data is unlikely to be able to violate security guarantees of an application, because attacker-controlled data is not of interest to the attacker by definition. Attacker-controlled data excludes in particular sensitive data, such as control-flow data, pointers, and cryptographic material used by the program.

In itself, memory categorization provides a looser but relevant form of memory safety: for instance, a use-after-free (UAF) of attacker-controlled data may only result in an access to other data categorized as attacker-controlled. Furthermore, memory categorization is well suited for being used as a mitigation-enabler, for achieving good security-performance tradeoffs. Once data is categorized as attacker-controlled, mitigations can be applied selectively to that data to achieve even stronger safety guaranties, while non-attacker-controlled data can be executed at native speed, without unnecessary mitigations.

We propose a memory categorization approach, referred to as MEMCAT herein, that *i*) is semi-automated: only sources of attacker-controlled data need to be specified, but no manual annotations of allocations are required to categorize memory, *ii*) has low overhead, *iii*) and categorizes both stack and heap data. MEMCAT builds on top of established and novel techniques and tools to provide additional allocators for stack and heap, categorize allocations based on their use, and apply this categorization.

We show two use cases, dropbear SSH and OpenSSL, where MEMCAT mitigates the exploitation of multiple severe vulnerabilities. In addition, in most cases MEMCAT causes low-to-negligible performance overhead with only 3% median overhead on the SPECint CPU 2006 benchmark. Nevertheless, some high overhead cases remain in our prototype implementation.

The main contributions of this paper are summarized as follows.

- We propose a new mitigation class, memory categorization, which separates attacker-controlled data from other data. If enforced thoroughly, memory categorization limits an attacker to reading or modifying solely its own data, drastically reducing its ability to compromise the system.
- We design MEMCAT, a low-overhead and automated approach to memory categorization. MEMCAT is based on static and dynamic analysis methods and applies memory categorization to both the program stack and heap.
- We implement MEMCAT on `x86-64` Linux and demonstrate its ability to mitigate past vulnerabilities on real-world software such as OpenSSL. We also evaluate its performance overhead on the SPECint 2006 benchmark suite.

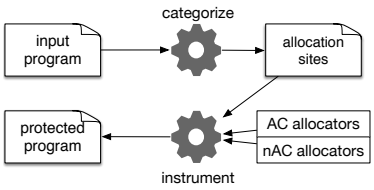

Fig. 1: A high-level overview of memory categorization. Given an input program, allocation sites are first identified and categorized. This categorization is then applied to the program by modifying the allocation behavior such that memory for (non) attacker-controlled data is only served from the corresponding allocators.

Threat Model. We assume an attacker that is capable of launching repeated attacks against a process with the goal of corrupting process memory. This includes attempts to both manipulate and disclose memory contents in an illegitimate way. Using the memory attack model introduced in [**memory:szekeres13**], this includes the ability to make a pointer go out of bounds, make a pointer become dangling and, as a consequence, use that pointer to write or read memory. MEMCAT is designed to thwart memory corruption of a process. It operates at the same privilege level as the victim program, assuming a benign OS.

2 Memory Categorization

The goal of memory categorization is to separate memory regions based on whether they contain attacker-controlled (AC) data or not (nAC). For instance, we separate network I/O buffers from data that is not directly attacker-controlled and thus trusted, such as control-flow data, pointers, and cryptographic material used by the program.

Three steps are required to achieve this goal (Figure 1): *i*) Provide mechanisms for separated memory allocation. We use separate allocators for both the heap and the stack to be able to serve memory for AC data from a different area than memory for nAC data. Allocators for AC data can additionally be hardened against memory corruption, using existing memory safety techniques. *ii*) Decide on a case-by-case basis from which region memory should be allocated. We use program analysis methods to identify which allocation sites, i.e. program locations that allocate memory, are used for AC data. *iii*) Implement this decision in the program. We use compile-time instrumentation and runtime support to apply the categorization result to the allocation behavior of the program.

We detail in the following each step.

2.1 Separate Allocation Mechanisms

A *memory region* is a contiguous chunk of memory that is provided by a *memory allocator*. The task of fulfilling memory allocation requests is generally satisfied by either the *stack* or the *heap*. The stack is a LIFO data structure that is composed of so-called frames, with frames being created upon function entry and removed on function exit. Allocating memory on the stack is very fast, allocating memory just requires moving the stack pointer (a dedicated register) to reserve a corresponding amount of stack space. In contrast, the *heap* is a large region of memory that is managed almost purely by a software support library. This software provides an interface to request variable-sized chunks of memory and releases them after use. While memory chunks provided by the

heap allocator can be accessed globally, heap allocation is generally more expensive than stack allocation.

MEMCAT intercepts both heap and stack allocations and extends them to properly handle attacker-controlled data to ensure that it is stored in a region that is separate from that used for nAC data.

Allocations on the stack are typically used only for a single purpose. As a consequence, MEMCAT only requires two separate stack allocators, one for AC and one for nAC data. Conversely, allocations on the heap are more likely to be used in more complex ways where a single memory region may store both AC and nAC data. Typical examples are more complex data structures such as linked lists, where the elements of the list store both list metadata and payload in a single allocation. As another example, some custom memory manager implementations use a single, large allocation that again hosts both payload and metadata. These use cases show that occasionally a single memory location may be used for disparate purposes. As a consequence, we introduce three heap allocators: one for AC data, one for nAC data, and one for allocations that mix nAC and AC data (referred to as *mixed*).

The mixed category remains prone to attacks: if a vulnerability related to attacker-controlled data categorized as mixed exists and if a sensitive piece of data was also categorized as mixed, then the attacker may succeed in an attack. Nevertheless, this category remains beneficial for multiple reasons: *i*) There may be no data of interest to the attacker in the mixed memory, rendering the vulnerability unexploitable in practice: e.g., in the case of an information leakage vulnerability such as Heartbleed, private keys will not be in the mixed category. *ii*) Categorizing mixed data as AC would be detrimental to security: it would make vulnerabilities corresponding to AC allocations exploitable (by targetting mixed data). *iii*) In practice, the set of allocations in mixed memory will be much lower than in nAC memory: this means that the mixed memory can be *selectively hardened* against attacks at low overall performance cost.

Allocation Sites. Memory allocators are invoked at locations in the program referred to as *allocation sites*. We identify allocations based on their allocation site to attribute them to a specific use. Different levels of detail are required for stack and heap allocations. Stack allocations are limited in scope, so the (static) program location in terms of calling function and offset is sufficient. For heap allocations, this is not sufficient: a program may invoke `malloc` from a single location to supply memory for all its components. This is precisely what happens with allocation wrappers, such as `xmalloc`, that decorate the functionality of the system's default heap allocator. If we were to use only the calling function and offset as an identifier we would conclude that there is only a single allocation site. This is why for heap allocation sites, we additionally require the full context of the calling function as a part of the allocation site identifier. In practice, the context is represented by the set of return addresses in the call stack that led to the allocation.

2.2 Allocation Decision

Different approaches are possible for deciding which memory allocator (AC, nAC or mixed) should be used at a specific allocation site. One approach is to let the program annotate variables (e.g., by annotating declarations with a special type). This approach is easy to implement, however it puts a high burden on the programmer and requires

modification of existing code. We opted for a more automated approach that only requires specifying AC data sources when they are used, e.g., specify any network receive or input parser function's buffer to be AC.

The choice of the memory allocator (AC, nAC or mixed) that must be used at a specific allocation site depends on how the allocated memory will be used later in the program. Allocators return a *pointer* that points to the allocated memory region. Pointers are used to both access the memory region and serve as a handle to it. In the course of program execution, these pointers may be modified and in turn be stored to memory. Our analysis process works as follows: we *i*) identify data sources that provide AC input, *ii*) track the pointers used to store data from these sources backwards to *iii*) find all allocation sites that have allocated memory for those pointers. We illustrate this process over the following code snippet:

```
1 char *cmalloc(int sz) {
2     if (sz == 0) return NULL;
3     return (char *)malloc(sz);
4 }
5 int main(int argc, char **argv) {
6     int fd = open(argv[1], O_RDONLY);
7     char *buf = cmalloc(10);
8     read(fd, buf, 10);
9     ...
10 }
```

In the beginning, MEMCAT identifies the `read` in line 8 as providing attacker-controlled input to the pointer `buf`. It then tracks back from line 8 to find the allocation site for `buf`, following the call to `cmalloc`, which leads to `malloc` in line 3 with the context being lines 7,3.

Note that the pointer itself is nAC - only the memory it points to is AC. As such, corruption of data in the mixed (or AC) heap cannot easily modify pointers.

Attacker-Controlled Input AC input is typically provided from data sources such as the network, hardware peripherals or storage systems (e.g., files). To obtain input from these sources, a program sends a service request to the operating system. The `read()` system call serves as a prime example. It reads potentially AC input from a file descriptor and places it in a memory region specified by a pointer argument. In practice memory categorization requires data sources to be identified for individual programs to be truly effective. At compile time, we walk through the source code and identify code regions that follow a similar semantic, i.e., writing AC input to a memory region, to flag the corresponding pointers as pointing to AC memory regions. In addition, we monitor these code regions at runtime to enhance the analysis if the compile time analysis was not conclusive for that case.

Allocation Sites Given a pointer that is associated with AC input, we need to identify its corresponding allocation site(s) to be able to change the allocation behavior. In the example above, the `read(fd, buf, 10)` matches the heap allocation site `malloc(sz)`, but in a different scenario it could also be a `char buf[32]` on the stack. We thus require precise information to determine which allocation site should provide what kind of memory given a specific invocation context. The precision of the information directly

affects the security benefits of memory categorization: While it will never worsen the security of an application, under-approximation makes it less effective as it potentially places AC data in memory supplied from nAC allocators. As our design assumes that AC allocators will use additional hardening (Section 3.3), we choose to err on the safe side and use over-approximating analysis methods to determine allocation sites of AC input. In this way, nAC data might be placed in memory supplied from an AC allocator. Because the AC allocator is hardened against exploitation, this will at most negatively impact performance, but not security.

Static Analysis To obtain precise analysis results at an interprocedural level for an entire program, we perform multiple analysis steps. We start with an intitial points-to analysis using Anderson's algorithm [**andersen94:programanalysis**]. Because Anderson's is context- and flow-insensitive, it can scale to a whole-program scope at an interprocedural level, while still operating at a field-level granularity. This precision is important for structs and classes with fields that point to both AC and nAC data – which would collapse to the mixed category otherwise. The result of this initial analysis is over-approximated points-to sets for every pointer used in the program. We feed these points-to sets to a static value-flow (SVF) analysis [**sui16:svf**]. SVF uses these points-to sets to construct an interprocedural memory single static assignment (MSSA) form of a program. The MSSA form of a progam extends the concept of single static assignment (SSA) for top-level variables to address-taken variables. When a pointer is dereferenced, i.e., an address-taken variable is loaded, this corresponds to a use of the address-taken variables the pointer refers to. When a pointer is assigned, i.e., an address-taken variable is stored, this corresponds to both a use and a def of the address-taken variables the pointer refers to. To capture interprocedural dependencies, callsites of functions that operate on address-taken variables of interprocedural scope also correspond to both a use and a def. The def-use chains of both top-level and address-taken variables are then used to create an interprocedural, sparse value flow graph (VFG) that connects the definition of each variable with its uses: In the VFG, nodes are either a definition of a variable at a non-call statement, a variable defined as a return value at a callsite, or a parameter defined at the entry of a procedure. The edges of the VFG represent the def-use value-flow dependencies, direct for top-level pointers, indirect for address-taken pointers.

In the next step, we focus on the pointers that are associated with AC input. We look up their nodes in the VFG. For each of them, we then perform a context-sensitive backward traversal on the VFG, adding precision on top of Andersen's points-to analysis for our pointers of interest. During the traversal we keep track of function return edges to construct the call stack that leads up to an allocation site, such that every time we reach a corresponding allocation site, we can now obtain the context under which the allocation site should actually provide a memory region for AC input.

Dynamic Analysis To complement static pointer analysis we intercept the functions that supply AC input also at runtime. We then detect which allocator – stack or heap – has been used for a given pointer by seeing where it fits with respect to the process memory layout. To obtain context information on heap allocations, we intercept heap allocators, unwind their call stack and associate it with the value of the pointer. While this information is only available after the allocation has happened, we need it to fill in potential information gaps of the static pointer analysis. Static pointer analysis is

limited to the code available at compile time, whereas at runtime programs dynamically link additional software components – shared libraries. Neither allocations nor uses of pointers within dynamically linked code are known to static analysis.

While our memory categorization approach focuses on a backward analysis, we also keep track on how pointers associated with AC memory regions are used during program execution. In particular, we investigate copy operations: If the source pointer is associated with a memory region tagged as AC or mixed, we also categorize the target memory region correspondingly. We deliberately keep this forward propagation of categorization results conservative, as research has shown that propagation policies that go beyond assignments lead to unconclusive results [**slowinska09:pointless**].

2.3 Changing Allocation Behavior

For the stack, compile-time analysis directly invokes the appropriate allocator based on whether the allocation is used for AC input or not. For the heap, compile-time analysis unwinds the call stack to determine the context in which an allocation site will provide a memory region used for AC input. At runtime, it adaptively changes the allocator's semantics based on the context information. When it encounters an allocation site for which it has no information, it serves the memory request from a separate data structure called the *limbo heap*. Write accesses to the limbo heap are intercepted by the runtime and analyzed based on whether they are associated with a data source supplying AC input or not. Once a memory region from the limbo heap is categorized as storing AC, nAC or mixed data, future allocations from the same site are served from the corresponding nAC, AC or mixed heap. MEMCAT also offers several heuristics for early categorization, detailed in Section 3.

2.4 Selective Hardening

In addition, the categorization also makes it possible to apply selective hardening efficiently, which has been shown to provide great performance improvements over full hardening even for costly mechanisms [**wagner15asap**, **kurmus14ccs**]. For MEMCAT, this simply means that the implementation of the nAC, AC, mixed heap or stacks can differ. In particular, previously costly hardened-heap mechanisms [**nikiforakis13:heapsentry**, **novark10:dieharder**, **akritidis10:cling**] can be applied to the AC (or mixed) heap, and only incur a modest performance overhead because only a fraction of all allocations will be redirected to these heaps. Assuming that the categorization does not misclassify an AC buffer into the nAC heap, this means that all the security benefits of the hardened heap can be provided at a fraction of its performance cost. Unlike the data-categorization-based approach of MEMCAT, the existing code-categorization-based approaches cannot be efficiently used with any existing memory safety mechanism. Indeed, many memory safety approaches require additional metadata to be tracked per data object: this means that a code-categorization-based approach will need to keep track of metadata for all objects, whereas a data-categorization-based approach can simply keep track of AC (or mixed) data's metadata.

3 Implementation

The implementation of MEMCAT consists of a compile-time and a runtime component. While providing the optimal protection together, both components can operate completely

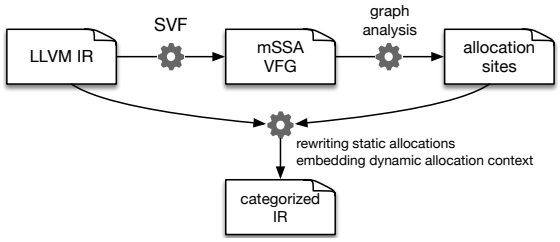

Fig. 2: Compile-time processing performed by MEMCAT. The LTO-LLVM IR representation of the program is first processed by SVF to build the memory-SSA form and the value-flow-graph. The graph is then traversed backwards from AC pointer use to the corresponding allocation sites. Static allocations are rewritten on the spot, for dynamic allocations the context information is embedded in the categorized IR output.

independently of one another. This lets MEMCAT provide protection even for programs where no source code is available.

3.1 Attacker-controlled data sources

Before categorizing memory in a program, MEMCAT requires configuration of the data sources considered to supply attacker-controlled data. An AC data source is specific to the program that is protected. In a simple program, I/O related functions such as `fgetc`, `fgets`, `fread`, `fscanf`, `pread`, `read`, `recv`, `recvfrom` and `recvmsg` are sources of AC data whenever they successfully write data into one of the buffers to be categorized. More complex software, which consists of multiple components that exchange data typically does so via interfaces with clear semantics that allow to specify whether a function supplies attacker-controlled data or not. In the case where a function can supply both AC or nAC data, MEMCAT's effectiveness can be significantly improved by providing a thin abstraction layer that separates these two cases.

3.2 Compile-Time

The main task of the MEMCAT compile-time component is to label all allocation sites based on what kind of memory should be provided. We implement the component as a compiler pass on the intermediate representation of the Clang/LLVM toolchain. To provide it with the most comprehensive view of the program, the pass works at the link-time-optimization (LTO) stage, in which all translation units of a program have already been linked together. Figure 2 shows an overview of the compile-time processing performed by MEMCAT.

The pass commences with Andersen's pointer analysis using a Wave [**pereira09:wave**] solver. It then uses sparse value flow analysis (SVF) [**sui16:svf**] to construct the mSSA form. The def-use chains for top-level pointers can be directly obtained from the LLVM IR, as it already is in SSA form, with one statement defining one variable. The semantics for address-taken pointers from the design section apply naturally to the LLVM IR's load and store instructions. To go interprocedural, entry and exit of functions are annotated with a def and use for non-local variables. These can then be linked to the arguments and return values at the callsites of a function. In the VFG, nodes are either statements (load, store, getelementptr), or parameters and return values. They are connected with intraprocedural or call/ret edges that can be either direct or indirect.

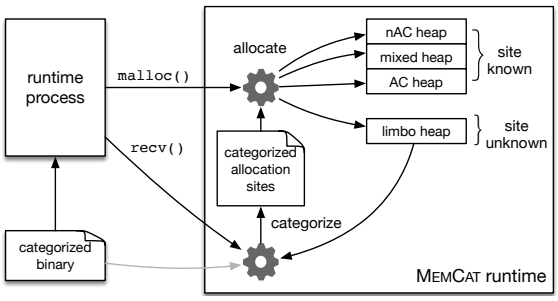

Fig. 3: Runtime activities by MEMCAT. Categorized allocation sites are read from the binary on startup. If an allocation from a known site is encountered, memory from the corresponding heap is served. If the allocation site is not known, memory from the limbo heap is served. Whenever AC data is written to memory, MEMCAT checks which limbo allocation the memory belongs to and categorizes the corresponding allocation site.

Then the pass iterates over the list of data sources that provide AC input. When MEMCAT finds a function invocation reflecting a data source from the list, it starts a backward traversal on the VFG starting from the pointer parameter supplied to the function. The backward traversal is done in a worklist-style manner, keeping track of the context. Context sensitivity is implemented based on associating every callsite with a unique ID, which also serves to prevent recursion on the VFG.

Whenever the backward traversal hits a heap allocation, we process the context of the allocation: To be able to refer to the return sites given by the context at runtime, we split the corresponding basic blocks at the return site and obtain their block address. To access this information at runtime, we add a global variable to the program executable that points to a two-dimensional array of all AC allocation site contexts.

Attacker-controlled stack allocations are replaced at compile time to point to offsets in the AC stack. The AC stack is implemented as an `mmap`ed memory region that can be accessed via two variables in thread-local storage that point to the base and the top of the stack. The implementation leverages LLVM's safe stack mechanism [**safestack**] and does not require further runtime support.

3.3 Runtime

The objectives of the heap runtime component are threefold: *i*) track all dynamic memory allocations, *ii*) categorize each allocation site *iii*) create secure boundaries between allocations that have been categorized differently. Figure 3 shows an overview of the runtime activities performed by MEMCAT.

Tracking We track memory allocations by intercepting all corresponding standard library calls used for allocation memory, such as `malloc`, `calloc` and `memalign`. Handling custom memory allocators is orthogonal to our work and has been addressed by its own line of research [**membrush:wcre2013**]. Glibc's hooks help applications override these functions by exposing the variable `__malloc_initialize_hook`, which points to a function that is called once when the heap implementation is initialized. This function can be overridden and used to overwrite four more function pointers (we refer to them as the *malloc hooks*) that are called by glibc before each heap-related function call.

Assigning identifiers to allocation sites To uniquely identify allocation sites we represent them with labels based on the call-stack contexts of the respective callsites of

the allocation. These identifiers are used both for the allocation sites categorized during compile- as well as runtime. Identifiers for each of the allocation sites (64-bit integers in our implementation) are obtained as follows: whenever one of the malloc hooks is called to allocate memory, we unwind the stack and extract return addresses for all the frames. The maximum call stack depth is a configurable parameter, which we set to twenty by default. We hash this array of return addresses onto a 64-bit integer as follows: initially the hash is zero. A loop dereferences each address in the array to obtain a quadword, which is circularly shifted by 7 positions, xored with the 12 least significant bits of the dereferenced address and finally xored with the hash. The hash is then circularly shifted by 13 positions to be ready for the next round. This approach ensures repeatable identifiers across executions despite the presence of randomized address spaces (e.g., ASLR): virtual addresses may be randomized, but *i*) the bytes they dereference remain constant by definition, and *ii*) randomization stops at a page granularity and so the 12 LSBs are bound to remain constant. Comparison with SHA256 over the entire array shows that the function has adequate collision resistance, with the highest collision rate registered at 0.2%.

Categorization The categorization process assigns a label to each allocation site identifier based on whether memory allocated at that allocation site is used to store AC, nAC, or mixed data, throughout its lifetime. Performing this determination is not trivial: at allocation time this information is not available to a purely runtime component. Therefore, it is necessary to hold off the categorization until the allocated memory is being used, because the categorization depends on the source of the data being stored.

A program is free to write to any allocated memory region any number of times at any offset, storing data coming from any source. Therefore, to categorize a buffer at runtime, it is necessary to keep tracking writes to it over the course of the execution of the program. To do this, we build a *limbo* heap that serves memory to all not-yet-categorized allocation sites. The limbo heap uses `mmap` to allocate one or more pages to satisfy the program's request for dynamic memory. The memory-mapped pages are `mprotect`'d to ensure that every attempt to write to them will generate a page fault. We implement a custom handler for `SIGSEGV` that behaves as follows: if the fault is not caused by any of the pages in the limbo heap, the program is terminated. Otherwise, write protection is removed from the page and the offending instruction (i.e. that which generated the page fault) is emulated. The emulation is performed by first decoding the instruction with udis86 [**udis86**] and by performing the required operation on the saved processor state (represented by the pointer to a `ucontext_t` struct provided as third argument by the operating system if the signal handler is registered with the `SA_SIGINFO` flag). The IP register saved in the context is then incremented to ensure that the offending instruction is skipped when execution resumes, and finally the protection on the page is re-introduced. This last step is required to keep tracking future writes to the page.

With this approach it is evident that a perfect categorization is an unreachable goal, given that usage of memory buffers might be data-dependent, and so it is always possible that the code makes a different use of the buffer in a future run. As a consequence, we develop heuristics to determine when the categorization can be declared complete. Note that until the categorization is complete, new allocations from the same allocation site have to be handled by the limbo heap, with the associated overhead. At the same time,

an early categorization might mistakenly assign a site to the wrong category. We have implemented and deployed the following heuristics: *i*) never stop the categorization process; *ii*) stop the categorization process after a configurable number of writes into the buffer; *iii*) stop as soon as all allocated bytes have been written to at least once. At the same time, functions such as `memset` and `bzero` do not affect the categorization process. This aims to capture the coding practice of zeroing out allocated buffers.

As soon as the categorization phase is declared complete for a given call site, the call site is labelled by associating the 64-bit call-site identifier with the integer representing one of the three labels. Whenever one of the malloc hooks is asked to allocate memory for the program, it determines the current call site identifier (as described above), searches whether a match is present in the map for that identifier and if so, allocates memory according to the label.

Handling allocations The heap runtime component includes a custom memory allocator that is based on ptmalloc2 from glibc. ptmalloc2 is a memory allocator where memory is served out of a pool of independent *arenas*. An arena is essentially a linked list of large, contiguous memory buffers obtained using `brk` or `mmap`. An arena is divided into *chunks* that are returned to the application. Ptmalloc2 uses a pool of arenas to minimize thread contention. Instead of a single pool of arenas, our custom allocator is designed and coded to handle three independent pools, one for each of the labels. The label of an allocation site serves as an additional argument transparently supplied by the heap runtime component to the custom allocator, indicating the pool that should supply the chunk to be given to the application. Call sites similarly labelled in the course of the categorization might be supplied with chunks from the same pool of arenas (potentially even the same arena, or the very same address). Conversely, call sites labelled differently are guaranteed to never receive addresses from the pool. Note that this guarantee needs to span across the entire lifetime of the program: for example, if the memory allocator releases the arena of a given pool (e.g. by calling `munmap`), it needs to guarantee that the same arena will not be reused for a different pool later.

In addition, to demonstrate the feasibility and low performance impact of selective hardening, we also implemented a hardened allocator solely for AC allocations. Since implementing the hardening mechanism is not the focus of this paper, we only chose to implement a simple hardened allocator and refer to related work (Section 6.2) for more elaborate heap hardening mechanisms. Our implementation is essentially an mmap-based allocator similar to PageHeap [**pageheap**] and ElectricFence [**efence**]: each memory allocation (including small allocations) is performed using the `mmap` call and is surrounded by guard pages. This mitigates many heap-related attacks by itself: uninitialized-data leaks are prevented because newly allocated pages are zeroed by the OS, heap-based buffer overflows (reads and writes) are prevented thanks to guard pages, and double-frees have no impact. Clearly, such an allocator would usually incur a prohibitive performance cost if all allocations were performed with this allocator (OpenBSD uses a similar allocator for their heap, but only for large zero-allocations for performance reasons [**moerbeek09malloc**]). However, MEMCAT will only categorize a fraction of heap allocations as attacker controlled, therefore, drastically reducing the performance overhead as shown in Section 4.

Categorization Propagation The categorization propagation component captures the case in which one of the identified AC inputs generates data into an intermediate buffer that is copied only later into the heap. It allows later copies into the heap to be categorized correctly as AC (or mixed).

The component hooks every function that supplies AC input. If the AC data *is not* copied into a buffer in the limbo heap, the component adds a *categorization record* into a global set. A categorization record is a tuple <`addr, len, start_ip, end_ip`>, where `addr` is the target address, `len` is the amount of data generated by this call to the AC data source, and `start_ip`, `end_ip` are the addresses of the first and last instruction of the caller of the AC data source function. Later, whenever an instruction is emulated as a result of a trap caused by a write into the limbo heap, we determine whether two conditions simultaneously hold: i) the source argument of the offending instruction draws input from a memory area that overlaps any categorization record's `addr, len` range and ii) the backtrace of the offending instruction shows that one of the return addresses is contained in the `start_ip, end_ip` range of the categorization record identified in the preceding condition. Informally, the second check determines whether the caller of the function supplying AC input is one of the callers of the function that is attempting to write into the limbo heap. This second check filters out false positives caused by one function writing AC data into a buffer, and a different function writing data into the limbo heap from the same address range used by the first function. Despite its simplicity, this component has proven capable of expanding the reach of the categorization process, as shown in Section 4.

Caching Any access to a buffer in the limbo heap will incur a high overhead because of the trap and subsequent emulation. This negative effect on performance is dampened by the heuristics for early categorization; however they are still not sufficient to achieve acceptable performance, because a limbo heap allocation might be long-lived; it may be the only one for that allocation site, or an allocation site can be visited several times before it is categorized.

To mitigate this problem, we have introduced a caching component to MEMCAT. This component persists to disk the data structure that maps allocation sites to labels across multiple runs. Whenever a program is restarted, the map of complete categorizations is loaded: any allocation site that was learned in a previous run will be directly handled by the appropriate allocator. Note that this is possible only because our hash function guarantees that call-site identifiers remain constant across executions.

4 Use Cases and Evaluation

We recall that in our threat model, inspired by [**memory:szekeres13**], the attacker aims to corrupt the memory of the process. This can be accomplished by making a pointer go out of bounds or by making a pointer become dangling. That pointer can later be used to write or read memory. By separating memory regions for AC and nAC data, MEMCAT mitigates exploitation of such memory corruption errors by design. With MEMCAT, pointers themselves are always nAC, as the address they point to is never supplied from an AC data source. The only, highly unlikely counter-example would be a program that reads a pointer address directly from standard input or the network. While pointers are

Vulnerability	Type	Program	Categorization	Mitigated?
CVE-2012-0920	use-after-free	Dropbear	AC	✓
CVE-2014-0160 (Heartbleed)	buffer overread	OpenSSL	mixed	✓
CVE-2016-6309	use-after-free	OpenSSL	AC	✓
CVE-2016-3189	use-after-free	bzip2	AC	✓

Table 1: Summary of analyzed vulnerabilities.

in general nAC, the memory regions they point to are potentially AC – the setting which we will discuss in the following.

We analyze the case of dangling pointers by focusing on how MEMCAT handles UAF vulnerabilities. With a UAF vulnerability, an attacker would normally exploit dangling pointers that reference memory that has already been freed. If the attacker can place data in the previously freed memory region, the program works with AC data whenever it accesses the dangling pointer. UAF vulnerabilities are even more crucial for exploitation of C++ applications, where objects are typically stored on the heap. In the case of objects that use virtual functions, the `vptr`, which points to an object's `vtable` that points to the virtual functions' addresses, is also stored on the heap. Here UAF vulnerabilities potentially allow an attacker to hijack the `vptr` with AC data – the entry point to execute hard-to-detect shellcode. MEMCAT mitigates exploitation of a UAF vulnerability as follows: if the affected pointer is nAC, the attacker will not be able to gain control over it as by definition data in a nAC pool may only be returned to a nAC allocation site. Conversely, if the affected pointer is AC, the attacker cannot read or modify anything that is not already attacker controlled data. This applies particularly to exploitation strategies that rely on hijacking the `vptr`, because C++ objects that do not have fields that are directly attacker-controlled are allocated on the nAC heap.

We now turn our attention to the second attack avenue considered in the threat model: out-of-bounds accesses to pointers. We first investigate the case of an out-of-bound read. This may, for instance, occur if the program is vulnerable to a heap-based overread and subsequent information on the heap is leaked. MEMCAT handles this by ensuring that the attacker can only cause an overread and leak data from "its own" heap, which typically means no sensitive information (in multi-user programs, there may still be sensitive information from other users, we discuss this limitation in Section 5).

For buffer overflows with write access, an attacker is essentially limited to overwriting data they already are in control of, effectively preventing classic buffer overflow attacks: indeed, in contrast to most fast heap implementations in use today, the AC heap separates heap metadata from actual heap content (selective hardening). In addition, each allocated chunk is isolated by guard pages.

4.1 Use Case: Dropbear

Dropbear SSH [**dropbear**] is a small SSH server that is designed for IoT devices such as WiFi routers or CCTV cameras. It can either run standalone or be compiled into utilities such as busybox [**busybox**]. As such it provides remote access and is typically accessible from the Internet. CVE-2012-0920 identifies a UAF vulnerability in Dropbear that allows for remote code execution. The affected memory location handled by the `char *forced_command` pointer field of the `struct PubKeyOptions` sets the command that a user who logs on with a key is limited to. By exploiting the UAF vulnerability, an attacker can remove this restriction to execute arbitrary commands.

Configured to categorize `read` invocation on network file descriptors as AC and read invocation on filesystem file descriptors as nAC, MEMCAT identifies a total of two

data source supplying AC input (namely, in `read_packet()` and `read_packet_init()`). It identifies a total of four heap and no stack allocation sites as being AC. When connecting with an SSH client to dropbear with MEMCAT, one of these allocation sites is actually encountered and memory from the AC heap is supplied.

On the first run, MEMCAT's runtime component allocates 305 times from the limbo heap. The allocations on the limbo heap result in 125,572 memory accesses that are intercepted. Based on the accesses to these allocations, MEMCAT moves three previously unknown allocation sites to the mixed heap. With categorization propagation enabled, this number increases to five allocation sites.

Because the forced command string is being read from the authorized keys file on disk, it is placed on the nAC heap. Therefore, the UAF cannot be exploited, because the AC data from the network is allocated on the separate, hardened, AC heap. Table 1 summarizes all vulnerabilities analyzed here.

4.2 Use Case: OpenSSL

OpenSSL [**openssl**] provides TLS, SSL and generic cryptographic support as both a shared library that is used by many programs and a standalone utility. In the standalone CLI tool, MEMCAT finds 22 data sources providing AC input and, based on this, categorizes 551 out of a total of 3648 stack allocations as AC. In terms of heap allocations, it categorizes 1724 allocation sites as AC.

To evaluate the standard runtime behavior of OpenSSL with MEMCAT, we run it in server mode and perform a TLS 1.2 handshake. On the first run, MEMCAT performs 1864 allocations from the limbo heap, resulting in 5,531,269 memory accesses that are intercepted. A total of six AC heap allocation sites are encountered during this test; one allocation site is categorized as mixed. The number of mixed allocation sites goes up to 36 with categorization propagation enabled. On the second run, MEMCAT leverages the cached allocation information from the first run and does not perform any limbo heap allocations.

To evaluate the performance overhead, we measure the time it takes to establish and shut down a TLS connection to OpenSSL running in server mode. Establishing and shutting down a connection performs all relevant operations, such as key agreement, hashing and (asymmetric) encryption, record parsing and I/O handling. The arithmetic mean of 100 measurements results in a 2.3% overhead, with 13.29*ms* vs 13.60*ms*. Examining the execution with perf [**perf**] using the CPU's performance counters, we saw that 87% of the time was spent in OpenSSL, 9.6% in our runtime, and 3.4% in the kernel. The kernel overhead is mostly caused by allocations on the limbo heap that cause traps into the kernel and to a lesser extent the system calls caused by allocations on the hardened heap. In the MEMCAT runtime, most time is spent on bookkeeping of the allocation sites as well as examination of the call stack. This data is obtained from 388 perf samples gathered during the duration of the 100 measurements. Note that the 9.6% of the time spent in our runtime include the time for all heap handling, which would normally be done by glibc. The overhead during regular data transfer is expected to be even lower, since only previously used ciphers are involved and I/O will be the limiting factor.

CVE-2016-6309 is a recent UAF vulnerability in OpenSSL where reallocation of the message-receive buffer through `realloc` potentially changes the buffer's location. This

	limbo	AC	mixed
1st handshake	1967	5	38
2nd handshake	4	5	39
1st heartbeat	20	5	40
2nd heartbeat	11	5	42

Table 2: Categorization results for OpenSSL in DTLS1 mode. We present numbers on the limbo allocation count and the number of allocation sites categorized as attacker-controlled and mixed.

is not reflected by the code, leaving dangling pointers to the old location. In this case, the UAF points to the AC heap. However, as the AC heap implementation in MEMCAT is additionally hardened against attacks, it is not exploitable.

Another, more prominent vulnerability is CVE-2014-0160, commonly known as Heartbleed. It is a memory disclosure bug caused by heap memory reuse: The heartbeat service routine of OpenSSL's DTLS1 protocol implementation allocates the buffer that is sent back to the client based on content parsed from incoming data. To be exact, the size of the allocated send buffer depends on a value of the receive buffer. The attacker can set this value such that a larger buffer than actually required is allocated, causing OpenSSL to send back uninitialized or, worse, reused memory content that has previously been freed. With MEMCAT, the incoming data is located in a buffer that is allocated from the AC heap. Executing the heartbeat routine the first time, MEMCAT has no information on the send buffer's allocation site and thus puts it on the limbo heap. During heartbeat processing, the send buffer is first initialized with a nAC padding. Then, data from the receive buffer is copied to the send buffer. MEMCAT's categorization propagation catches this and changes the categorization of the send buffer's allocation site from nAC to mixed. All subsequent heartbeat responses are then allocated from the mixed heap. Table 2 shows the categorization progress across execution of the handshake and heartbeat code. Using a hardened heap implementation (such as the mmap based one we use in the AC heap) for the mixed heap would mitigate the exploitation of this vulnerability. Without it, however, the impact is also reduced, because an attacker can only leak data from the mixed heap, reducing the attack surface by a factor of 46 in terms of allocation sites, with 1952 allocation sites categorized as nAC and 42 as mixed.

4.3 Performance

To evaluate MEMCAT's impact on performance, we use SPECintCPU 2006 [**spec**], a standardized, well-established benchmark suite, with no exceptions. We do not run SPECfp CPU 2006 benchmarks such as namd because we did not deem floating point benchmarks to be affected by any MEMCAT changes. We perform all experiments on Ubuntu 14.04 running on an Intel(R) Core(TM) i7-6700K CPU clocked at 4 GHz with 32 GB memory. For the performance evaluation, we configured MEMCAT to use `(f)read`, `recv(from)` and `(f)gets` as data sources providing AC input. We report the runtime categorization results from the last run of every benchmark. The allocation sites actually encountered at runtime are fewer than the ones categorized at compile time, because the actual execution of the benchmarks covers only a subset of the code paths.

Table 3 shows the categorization results. 462.libquantum does not use any of the configured data sources and thus no stack or heap allocations are categorized as AC. For 483.xalancbmk, the points-to analysis was not able to obtain a points-to set for the

Benchmark	compile time AC data		runtime heap allocations			
	input	stack	heap	nAC	mixed	AC
perlbench	7	124	31	9185	0	15
bzip2	1	0	3	9	0	3
gcc	4	2	5	266404	1	0
mcf	3	1	1	6	0	0
gobmk	10	5	1	3672	0	0
hmmer	119	38	2525	83	1	65
sjeng	5	2	1	5	0	0
libquantum	0	0	0	7	0	0
h264ref	4	0	2	157	1	2
omnetpp	6	2	2	10305	0	0
astar	27	2	4	181	0	3
xalancbmk	1	0	0	4832	3	0

Table 3: Categorization results for the SPEC benchmark programs. The number of data sources supplying AC input and the number of corresponding stack and heap allocations categorized as AC at compile time. The number of allocation sites for nAC, mixed and AC heap encountered at runtime.

pointer associated with the AC input data source. Interestingly, 401.bzip2 is actually susceptible to CVE-2016-3189, a UAF vulnerability mitigated by MEMCAT.

Figure 4 shows the runtime overhead of MEMCAT, the baseline being the SPECintCPU 2006 suite compiled with Clang/LLVM with LTO. The stack categorization, which is largely a compile-time effort, only incurs an average (geometric mean) overhead of 0.1% during runtime. Categorization of heap allocations, which requires more runtime support has a higher average overhead of 21%. This remains a reasonable overhead: for example, the DieHarder [**novark10:dieharder**] hardened heap implementation, which is much more efficiently implemented than our simple mmap-based heap implementation used in the AC heap, incurs on average 30-40% overhead over the glibc heap allocator (which we also use in our baseline), also on SPECintCPU 2006. Of course, this is because DieHarder replaces all heap allocations, whereas MEMCAT only does so selectively where it matters most. This demonstrates that MEMCAT can be successfully combined with approaches such as DieHarder, to further reduce their performance overhead in practice and make them practical to use.

Although some benchmark results such as xalancbmk show a high overhead, SPECintCPU 2006 is known to highly stress the memory allocator on some benchmarks: Table 4 gives the per-benchmark details on how often the allocator is invoked. The overhead incurred by MEMCAT is incurred when a memory region is allocated, when access to a memory region on the limbo heap is intercepted, or when the slower AC heap is used. As can be seen, we experience a higher performance overhead on the benchmarks that perform an exceptionally high number of allocations, as many as over 267 million in the case of omnetpp. We also include the observations of the call stack depth. Even though we enforce a depth limit of twenty frames when examining the call stack to extract the context for allocation sites, constantly deep call stacks as caused by xalancmbk's recursive programming, require MEMCAT to always unwind until the limit.

5 Discussion and Future Work

Inaccurate categorization. The most pressing question with respect to MEMCAT is the potential impact of an inaccurate categorization. In general, the security guarantees provided are commensurate to the accuracy of the categorization. False positives may

Benchmark	allocation count	call stack depth
400.perlbench	$> 56M$	> 60
401.bzip2	28	4
403.gcc	$> 2.9M$	> 50
429.mcf	3	4
445.gobmk	$> 118K$	> 20
456.hmmer	$> 1M$	6
458.sjeng	4	4
462.libquantum	179	9
464.h264ref	5683	10

Table 4: Heap allocatior `realloc` and `calloc` to measure the stress on the n allocation site.

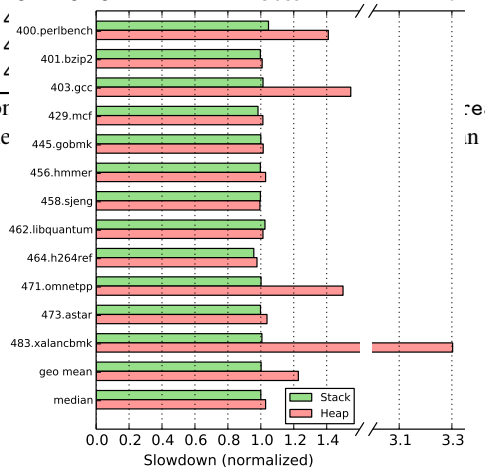

Fig. 4: SPECint CPU 2006 benchmark overhead.

result in nAC data allocated in the AC heap or stack. While this can degrade security if the nAC data is sensitive, the impact is mitigated by additional hardening mechanisms that we apply to the AC memory regions. False negatives instead occur when AC data is allocated in a nAC memory region. This has no consequences as long as the buffer holding AC data is not part of an actual vulnerability.

Sensitive AC data. There is still one aspect that is not captured by the current implementation of MEMCAT: not all attacker-controlled data may be unimportant for the attacker, for instance, in a program with multiple distrusting users. If the program receives data from multiple distrusting external sources, the categorization process may conclude that they are all attacker controlled, whereas the attacker only controls a subset of them and might find it advantageous to learn or modify others. An extension to MEMCAT could cater for this fact by implementing more than one AC label, one for each distinct external untrusted data source, isolating data of one user from that of another. However, this would require associating data sources with the distinct external sources.

Propagating Categorization. Another question that might arise is whether we "miss" AC data in memory regions because of our conservative categorization propagation method. We acknowledge that more thorough dynamic taint analysis methods could be used that handle, for example, implicit data flows [**dta:kang11**]. However, this raises yet another question: how far should we go in propagating categorization results? Based on

previous research [**slowinska09:pointless**] on taint explosion and on the positive results we obtained with the current implementation, we deliberately chose to limit the scope of propagating categorization results to memory copy operations.

6 Related Work

Increasing memory safety for C and C++ programs for the sake of security, keeping overhead and complexity low, has been the aim of a large body of research tackling the problem from different angles.

6.1 Memory Integrity

Control flow integrity (CFI) [**cfi:abadi05**] is the property that is violated if a program's execution deviates from its intended flow. If it is measured and enforced to full extent, attacks that corrupt code pointers to hijack control flow can be prevented completely. Apart from numerous limitations and challenges in practice [**cfbending:calini15**, **overcoming˙cfi:goktas14**], this approach does not address attacks that solely rely on modifying non-control data [**coop:schuster15**]. The same drawback applies to code pointer integrity [**cpi:kuznetsov14**], which hides code pointers from being accessed by an attacker by storing them in a separate memory region. Write integrity testing [**akritidis08:wit**] aims at preventing memory corruption in general. For every memory write instruction, it computes the set of objects that it can legitimately write to. During runtime, it enforces that write accesses are limited to their legitimate object's memory region. While this approach addresses the corruption of non-control data, it does not take memory reads into account and thus does not protect against memory disclosure. Data flow integrity [**castro06:dfi**] uses reaching-definition analysis to ensure that only legitimate instructions – the ones in the definition set – have written to memory locations at the time of use. This requires keeping track of the last write instruction for every memory location, incurring a significant overhead. In contrast, MEMCAT does not try to achieve memory integrity in general, but rather separates AC data such that attackers are limited to corrupting their own data. At the same time, separation allows performance-intensive integrity checks to focus on AC data, not affecting nAC data.

LLVM's SafeStack [**safestack**] implements a dedicated stack memory area for address-taken stack variables, separating the latter from return addresses and register spills to avoid stack corruption. MEMCAT goes further by taking into account whether a variable holds attacker-controlled data, instead of blindly separating all address-taken variables. In contrast to SafeStack, with MEMCAT, an AC address-taken variable cannot overflow into a nAC address-taken variable. Also, nAC address-taken variables will not be put on a separate stack, leading to lower overhead. Finally, as the name already implies, SafeStack does not address data residing on the heap.

Finally DataShield [**datashield:carr17**] allows the programmer to annotate types as sensitive. It separates memory in isolated sensitive and non-sensitive areas, albeit with only partial support for the stack, to prevent non-control-data attacks with traditional memory safety checks when accessing sensitive data. Unlike MEMCAT, it requires manual modification of the source code by the programmer (automating this work is a large part of the static analysis and runtime components of MEMCAT), and is designed to categorize control-flow-data as non-sensitive.

6.2 Heap Protection

Efforts in making the heap more resilient against attacks include Cling [**akritidis10:cling**], dieharder [**novark10:dieharder**] and HeapSentry [**nikiforakis13:heapsentry**]. Cling mitigates UAF attacks by ensuring type-safe memory reuse, such that heap memory chunks can only be recycled for objects of the same type. DieHarder [**novark10:dieharder**] combines a multiple number of hardening mechanisms, some of them borrowed from OpenBSD's heap allocator. Among them is full segregation of heap metadata from normal data to mitigate metadata corruption and randomized heap space reuse. HeapSentry adds kernel-enforced canaries to allocated memory regions to prevent heap buffer overflows. These approaches harden the heap, which is orthogonal to what MEMCAT strives to achieve. However, MEMCAT can certainly benefit from these approaches if deployed on top: they can add additional protection to the AC and the mixed heap, keeping the overhead on the nAC heap low.

Mitigating UAF vulnerabilities through pointer tracking has been implemented in Dangnull [**lee15:dangnull**] and FreeSentry [**yves15:freesentry**]. These systems set all pointers that refer to memory to null once the corresponding memory region is freed. While being effective in preventing UAF from being exploited, they only address a single class of vulnerabilities at a comparatively high – 80% average on SPEC – overhead.

Microsoft's Isolated Heap feature introduced in 2014 [**isoheap**] is a UAF mitigation feature for Internet Explorer. It comes closest to our work and implements a form of memory categorization, limited only to the heap, and for a manually selected set of objects. A similar mitigation has been ported to Adobe's Flash player [**flashisoheap**]. Inline with our evaluation, the Isolated heap feature has been recognized as a highly useful mitigation technique [**pzeroheap**]. In comparison to these existing approaches, MEMCAT provides semi-automatic categorization of objects through static and dynamic allocation, which makes the technique more broadly applicable, and potentially less error prone.

6.3 Taint Propagation

Dynamic taint analysis methods [**qin06:lift**, **kemerlis12:libdft**, **minemu:bosman2011**] track the data flow of taint labels throughout program execution. In literature, taint analysis has been promoted for vulnerability detection in the past: Taint labels are typically assigned at pre-identified data sources, and an error is raised if tainted data reaches certain sinks, for example, input from `read()` is used in `execve()`. The dynamic tracking of the taint requires instrumenting the execution of a given program, typically incurring a significant overhead unless implemented leveraging CPU features [**minemu:bosman2011**]. Implicit data flows that potentially hinder taint propagation can be addressed using heuristics if source code is available [**dta:kang11**].

While taint analysis is not among the main aspects of how MEMCAT mitigates memory corruption, its categorization propagation method to identify AC memory regions beyond an AC data source can be extended using the methods mentioned above.

7 Conclusion

In this paper, we present *memory categorization*, a mitigation for memory corruption attacks. Memory categorization analyzes and labels memory allocation sites and enforces the separation of attacker-controlled. It follows up on the successful isolated

heap [**pzeroheap**, **flashisoheap**] feature used in practice by Microsoft and Adobe, by introducing the concept of memory categorization in general and semi-automizing the categorization process itself.

In itself, memory categorization mitigates memory corruption vulnerabilities, such as buffer overruns or dangling pointers (e.g. use after free vulnerabilities). Our evaluation on real-world vulnerabilities in Dropbear and OpenSSL demonstrates the effectiveness of MEMCAT, while our performance evaluation shows that it comes at little cost.

Furthermore, this approach can be extended along two lines: *i*) targeted hardening of the allocators supplying memory over which the attacker has full or partial control as well as *ii*) permitting the selective use of otherwise impractical tools or techniques that provide full memory safety based on memory categorization.