PUMM: Preventing Use-After-Free Using Execution Unit Partitioning

Carter Yagemann
The Ohio State University[*]

Simon P. Chung
Georgia Institute of Technology

Brendan Saltaformaggio
Georgia Institute of Technology

Wenke Lee
Georgia Institute of Technology

Abstract

Critical software is written in memory unsafe languages that are vulnerable to use-after-free and double free bugs. This has led to proposals to secure memory allocators by strategically deferring memory reallocations long enough to make such bugs unexploitable. Unfortunately, existing solutions suffer from high runtime and memory overheads. Seeking a better solution, we propose to profile programs to identify units of code that correspond to the handling of individual tasks. With the intuition that little to no data should flow between separate tasks at runtime, reallocation of memory freed by the currently executing unit is deferred until after its completion; just long enough to prevent use-after-free exploitation.

To demonstrate the efficacy of our design, we implement a prototype for Linux, PUMM, which consists of an offline profiler and an online enforcer that transparently wraps standard libraries to protect C/C++ binaries. In our evaluation of 40 real-world and 3,000 synthetic vulnerabilities across 26 programs, including complex multi-threaded cases like the Chakra JavaScript engine, PUMM successfully thwarts all real-world exploits, and only allows 4 synthetic exploits, while reducing memory overhead by 52.0% over prior work and incurring an average runtime overhead of 2.04%.

1 Introduction

Memory unsafe languages like C and C++ are pervasive and contain difficult to spot binary level bugs that attackers regularly exploit. One of the most prevalent and tricky to detect classes is use-after-free (UAF), which occurs when a program frees dynamically allocated memory, but then later uses a dangling pointer, causing an unsafe memory access. This includes another prevalent class, double free, where the violating use is a second call to free an already freed buffer. In the mildest cases, UAF can lead to a program crash and *denial of service* (DoS), but more often arbitrary code execution becomes possible, enabling an attacker to take complete control of the program. In 2020 alone, NIST published **73** UAF advisories for Google Chrome and **17** for Mozilla Firefox. UAF has enabled arbitrary code execution in Chrome [19], Adobe Reader [23], and Windows 10 [18], to name a few.

The fundamental factor that elevates UAF from causing DoS to enabling arbitrary code execution is the attacker's ability to control what data gets reallocated to the newly freed address space. For example, if memory containing a code pointer is freed and then reallocated and overwritten with a buffer whose contents are chosen by the attacker, triggering a UAF will cause the program to deference this buffer as if it were still the original code pointer, sending the program counter to whichever address the attacker chooses.

In response, researchers have proposed to redesign memory allocators to defer reallocation just long enough to prevent exploitation. The intuition is that if the attacker cannot overwrite prior memory objects with new ones (as in the previous example), then code execution will no longer be possible via UAF. Viable solutions need to offer a low overhead, drop-in replacement for standard management libraries (e.g., libc) and be able to function in binary-only settings (without access to source code or debug symbols) to retrofit protection onto *commercial off-the-shelf* (COTS) and legacy software.

In recent years, two strategies have emerged to solve this problem: scanning and *one-time allocation* (OTA).[1] In the scanning approach, freed addresses are quarantined until a scan of memory verifies that no possible pointers remain, at which point they are released for reallocation [1]. Conceptually, scanning is similar to retrofitting garbage collection into program binaries. Conversely, the OTA approach enforces that an allocated address will never be reallocated again during the program's lifetime [61].

Both approaches offer straightforward security guarantees, however they still suffer practical and fundamental limitations. Scanning does not work on code that obfuscates pointers [9] (e.g., by combining them with flags or reference counters)

[*]Work done while at the Georgia Institute of Technology.

[1]There are also approaches based on "fat" pointers [30], but these require source code or dynamic instrumentation and incur significant overhead.

and may excessively quarantine memory due to false positive detections. Conversely, OTA will eventually exhaust the program's virtual address space since no reallocations are allowed. In both cases, the current best-of-breed implementations still incur worst-case execution and memory overheads of 50% and 100%, respectively.

Seeking a better solution, we turn our attention to another area of research, data provenance [31], which has spent years tackling a seemingly orthogonal problem: false provenance [58] (a.k.a. false dependency explosion). Here, researchers aim to causally link subjects (processes) to objects (sockets, files, etc.) based on system event logs (system calls [25], application log messages [26]) to facilitate attack detection and forensic investigation. However, they quickly discovered that long running programs incur false dependencies, whereby an object is wrongly determined to be causally related to every prior object the program touched. To address this, researchers have refined a technique called execution unit partitioning (EUP), which dices the program into *autonomous units of work* to partition dependencies [33, 40]. For example, a unit for a HTTP server consists of the code that processes one request, which the server iterates until termination.

What makes EUP interesting is that it works because units carry the special property that *subsequent iterations of a unit have no data dependencies to prior iterations*. This is what enables them to accurately partition provenance without pruning relevant events. From this observation, we formulate the hypothesis that *execution units can also serve as an effective guide for when freed memory is safe to reallocate.* For example, once a HTTP server moves on to the next request, it is unlikely to access any of the prior request's pointers, even if some were left dangling in memory, and any newly allocated data will be initialized, safely overwriting old values. While this is a heuristic that cannot be guaranteed to hold in all cases, its empirical robustness has been demonstrated over a decade of formal research [5, 24, 29, 31, 33, 35, 37–41, 60].

Conceptually, our idea can be thought of as enforcing OTA *at the granularity of execution units* rather than globally. Since units can be identified offline, policies based on them are efficiently enforceable at runtime without requiring CPU-intensive scanning. Since quarantined addresses are guaranteed to be released at the start of each new iteration, the address space will not be exhaustible like in global OTA. In short, our strategy avoids both the performance overhead of scanning and the memory overhead of OTA.

However, turning this idea into a working design is not a trivial task. The first challenge we have to overcome is how to identify suitable units for guiding deferred reallocation in arbitrary programs that may be stripped and lacking source code. To this end, we propose a technique based on offline dynamic profiling to detect the outer loop indicative of an execution unit, which is a common pattern heavily validated and trusted by the data provenance community [26, 31, 33, 39, 40]. Our analysis is compatible with dynamic instrumentation (e.g., DynamoRIO [22]) or hardware processor tracing (PT) provided by mature frameworks like Perf, achieving wide compatibility across modern devices.

Notice that using low-level runtime behaviors, rather than audit logs, to detect units is a unique departure from prior work that reflects our goal of managing memory rather than partitioning system calls and application messages. Consequently, while EUP is a good starting inspiration, the algorithm we propose is unique in the granularity of the units it identifies and is not a direct transplant of prior EUP work.

Next, we shift our attention to turning identified units into efficiently enforceable quarantine policies for memory allocators. To this end, we propose an algorithm to locate release points in the program that can be accurately identified at runtime without requiring any added instrumentation or PT. Specifically, our design analyzes the control flow of the profiled traces to identify callers to the memory manager's functions that occur at the beginning of known execution units. These callers are easy to identify at runtime by the return pointer pushed onto the stack, allowing for safe releasing of prior quarantined addresses at the start of each new iteration of the unit.

We have implemented a prototype of our design for Linux, PUMM,[2] supporting profiling data collected from Perf using Intel PT, enforced at runtime using a drop-in wrapper for libc. In an evaluation exploiting 40 unique real-world and 3,000 synthetically generated vulnerabilities in 26 commodity programs, including complex software like the Chakra JavaScript engine, PUMM prevents all the real-world UAFs and all but 4 synthetic UAFs from being exploited while incurring a 2.04% execution and 16.5% memory overhead, on average. Compared to the current best defenses based on scanning and OTA (MarkUs [1] and FFmalloc [61]), our approach reduces execution and memory overheads by 2.74% and 52.0%. On the SPEC CPU2006 benchmark, our system also excels with execution and memory overheads of 3.12% and 1.87%. We manually verify the correctness of the real-world units identified by PUMM and empirically validate its robustness to varying code coverage during profiling. We have released the code for our prototype to facilitate future work.[3]

2 Overview

2.1 Motivating Example

To demonstrate concretely how PUMM works, consider the example shown in Figure 1, shown as source code for clarity (PUMM's actual analysis only requires the program binary). For simplicity, we have reduce this example to just the file parsing routine of the program.

In this example, the function `getline` is imported and calls `realloc` on line 16 to dynamically reallocate buffers

[2] Partitioned Unit Memory Management.

[3] https://github.com/carter-yagemann/PUMM

```
1. void parse(char *line) {
2.   int counts[26] = {0};
3.   char *ptr = line;
4.   int idx;
5.   while (*ptr) {/*...*/}
6. }

7. void foobar(FILE *i) {
8.  char *r = NULL;
9.  size_t s = 0;
10.  while(getline(&r,&s,i))
11.      parse(r);
12.  free(r);
13. }

14. int getline(char **r,
       size_t *s, FILE *i) {
15.   *s = next_s(i);
16.   *r = realloc(r, *s);
17.   /*...*/
18. }
```

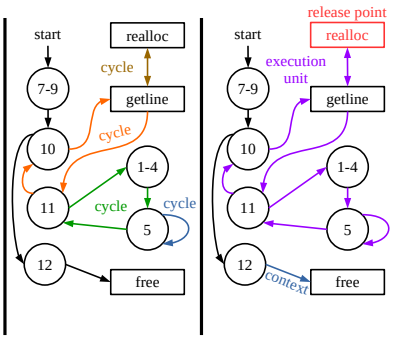

Figure 1: Motivating example. PUMM's profiling reveals 4 simple cycles, which are merged into 1 unit (purple). Once `realloc` is reached, none of the pointers previously freed by `getline` are used, so they can safely be released.

for storing variable length lines. These are then returned to the routine as pointers. Unfortunately, line 10 allows for a UAF vulnerability because it does not correctly check the value returned by `getline`, which may be negative to indicate an error, meaning line 11 may pass `parse` a dangling pointer.

How does PUMM prevent this bug from being exploited? First, during the offline phase, PUMM profiles the program to collect execution traces of test inputs, revealing the shown control flow graph (CFG). In this case, PUMM detects 4 simple cycles (a.k.a. elementary circuits) and since the head of the orange cycle (Node 10) dominates the heads of the other cycles, they are merged together into 1 execution unit, shown in purple. Notice that the head of this unit (Node 10) is also the head of the code's outermost loop. This is not a coincidence and will be elaborated on in Subsection 3.2, but for now, observe that the resulting unit has 1 entrance/exit node corresponding to the while condition on line 10. Also notice that any pointers to memory dynamically allocated by `getline` are initialized at the start of an iteration and never accessed in subsequent iterations. This makes sense because the unit we have identified contains all the code to autonomously process one input (i.e., one file line).

With all the units identified, PUMM next locates all the callers to allocating memory manager functions and checks whether any belong to a unit. In this example, there is a call to `realloc` near the head of the unit. Notice that reaching this call indicates the start of a new iteration of the unit and once reached, no data from prior iterations is accessed. Consequently, any dangling pointers prior iterations may have left in memory will not be used again, posing no UAF risk. PUMM records the caller of `realloc` in its security profile as a safe caller for releasing quarantined addresses.

At runtime, PUMM's memory management wrapper is loaded and linked by the OS. For Linux, this can be configured in `ld`'s settings or performed with the `LD_PRELOAD` environment variable. In either case, PUMM's wrapper detects that the program has a security profile and loads it. Each time `getline` calls `realloc`, any memory freed by the call is quarantined and previously quarantined memory is released for reallocation. Notice that under this policy, it is still possible for the UAF bug to trigger a crash. However, because the improperly accessed memory is quarantined, the UAF can only access the originally freed data, making it unexploitable for attackers. Subsection 2.2 elaborates on the taxonomy of UAF exploits and Subsection 2.3 defines our threat model.

	Memory State	Access	Exploitable?
L1	Inaccessible	Crash	No
L2	Never Reallocated	Old Data	No*
L3	Reallocated, Uncontrolled	New Data	Maybe
L4	Reallocated, Controlled	Attacker's Data	Yes

* For secure memory allocators.

Table 1: Consequences of use-after-free.

2.2 Use-After-Free Exploitation

The consequence of having a UAF bug within a program varies substantially depending on the state of accessed memory, summarized in Table 1. In the mildest case (L1), the dangling pointer left in memory references an address that has been unmapped or released, making it no longer accessible to the program. This can cause a crash, or create a DoS scenario if the UAF is triggered repeatably, but it is not exploitable to control the program's execution.

At L2, the freed memory referenced by the pointer is still accessible, but has not been reallocated, so it still contains the original data prior to being freed. This can cause a crash or yield no observable anomaly, depending on the program's logic, and is no more exploitable than L1 for secure allocators. Conversely, for allocators that overwrite freed memory with metadata, it may be possible to achieve exploitation [52], however this is unlikely and can be avoided by changing where metadata is stored in the memory manager's code.

L3 and L4 are when UAF becomes a serious threat. In these cases, the freed memory has been reallocated, causing the old data to be overwritten with new data. If an old code pointer or control-dependent variable is overwritten, the UAF can redirect program execution, achieving exploitation. When the new data is not controlled by the attacker (L3), success depends on the probability of the program overwriting the old data with values that benefit the attacker. Conversely, if the attacker is able to control what new values are written (L4), arbitrary code execution is achieved.

2.3 Threat Model

Similar to prior work [1, 61], our goal is to prevent UAF from being exploitable. This means that an attacker should not be able to control what behavior the program exhibits by triggering a UAF bug (i.e., preventing L3 and L4 in Subsection 2.2).

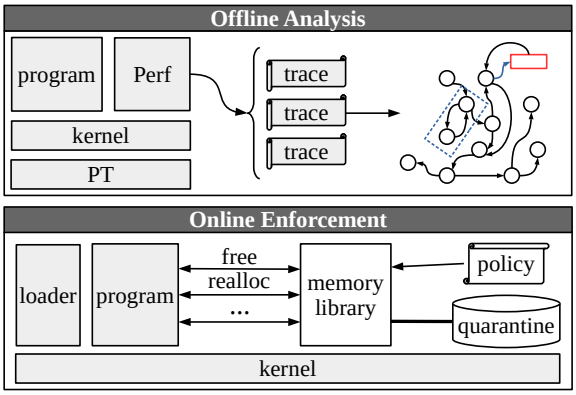

Figure 2: System architecture. PUMM consists of an offline analysis phase and an online enforcer that wraps the program's memory management library.

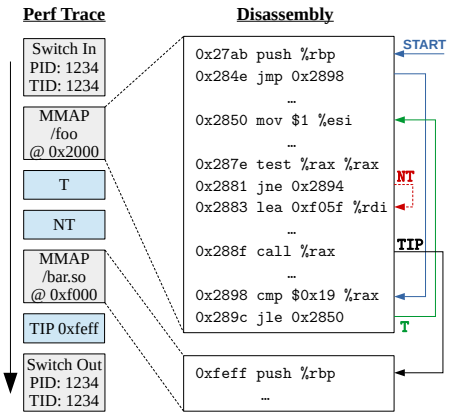

Figure 3: Example of PUMM disassembling an execution trace using the recorded PT data and Perf sideband.

Under this definition, crashing at the UAF (L1) or accessing old data and continuing (L2) is satisfactory because no harm is done to the system, assuming the allocator does not store metadata in freed memory. However, for completeness, we distinguish between these outcomes in our evaluation because crashing can still lead to DoS.

We assume that the program being targeted starts in an uncompromised state, motivating the attacker's desire to launch a binary exploit. The adversary has full knowledge of how the target program works and what UAF bugs it contains. This includes being able to trigger arbitrary reads and data leaks to reveal secrets (e.g., stack canaries, ASLR offsets) to help craft a working exploit. Since our policy is enforced at runtime using a loaded library, the attacker can also read it. However, we assume the attacker does not have an arbitrary write capability, otherwise they could rewrite freed data while in quarantine to achieve exploitation. Similarly, we do not consider an attacker that can arbitrarily create new program paths. These limitations apply to all the prior work [1, 32, 34, 42, 53, 55, 61, 63] and are reasonable because arbitrary write and control flow hijacking are stronger capabilities. In other words, if an attacker already has either of these, they do not need to exploit a UAF because they can already redirect execution [7, 8, 10, 14, 28, 51, 57].

We assume that the kernel of the targeted system is uncompromised because we rely on it to load our enforcement code, as does prior work [1, 32, 34, 42, 53, 61]. This is realistic because if the attacker already controls the kernel, they control the system and do not need to exploit user program bugs. We also assume the security profile can be securely stored and loaded by our library, which is achieved in practice with file permissions.

3 Design

PUMM consists of an offline binary analysis phase that yields a security profile that is then enforced at runtime using a memory management wrapper, as shown in Figure 2. Profiling for the offline phase can be facilitated using either dynamic binary instrumentation [22] or PT. For brevity, we only describe using Linux Perf and Intel PT, but keep in mind that instrumented code can emulate the exact same functionality.

First, PUMM records execution traces of the program handling test inputs and decodes them into a CFG (Subsection 3.1). PUMM then uses this CFG to perform an analysis that identifies the execution units within the binary code (Subsection 3.2). Finally, PUMM identifies places in the program where quarantined memory addresses can be safely released for reallocation, yielding a security policy for the runtime wrapper (Subsection 3.3).

3.1 Dynamic Profiling

Before PUMM can create a security policy, it first needs to know what execution paths exist in the target program. Since we need to handle the case where no source code is available to support legacy and COTS programs, PUMM cannot rely on accurate compiler-based approaches like those offered by LLVM [11]. Conversely, static binary analysis is difficult to perform, especially on obfuscated programs, and does not scale to large complex code [54]. Instead, PUMM relies on dynamic profiling to record execution traces of the target program. For the purposes of this work, we collect inputs to drive the profiling phase using a fuzzer seeded with developer test cases. The impact of code coverage is measured in our evaluation.

Processor Tracing Our prototype for PUMM utilizes Intel PT, which is a hardware feature first introduced in Skylake

processors, and Perf to record execution traces. From Perf's trace data, PUMM can recover each instruction the program executed, even in complicated scenarios involving multiple threads and dynamically generated code.

Figure 3 shows an example of the recovery process. On the left is the recorded trace, encoded as a stream of Intel PT packets (blue) and Perf sideband packets (grey). For brevity, the figure does not display all the data contained in each packet (e.g., CPU timestamps). The first packet informs PUMM that the target thread (PID 1234) has started executing on a processor core. This is followed by a `mmap` event that tells PUMM that an object was loaded from storage into memory. The values record which object it was and the base virtual address, allowing PUMM to recreate the same memory layout within its disassembler. The next packet contains PT data, so PUMM resumes linearly disassembling instructions until it reaches a branch (`jle`). It then consults the PT packet, which says the branch was taken, so PUMM jumps to the appropriate target address (0x2850) and resumes. At the next branch (`jne`), the next PT packet records not-taken, so PUMM falls through to the next instruction (0x2883). The next packet is another `mmap` event, so PUMM loads the specified object. When it reaches the indirect call at 0x288f, the next PT packet contains the target IP (TIP), directing the disassembler to 0xfeff. Continuing this procedure, PUMM recovers every executed instruction.

Notice that due to address space layout randomization (ASLR), the virtual addresses of instructions change with each execution. To account for this, PUMM converts the absolute virtual addresses into offsets relative to object bases. This allows subsequent steps in PUMM's analysis to merge traces into a single unified graph.

Control Flow Graph Construction Once PUMM has recovered the instructions executed in the trace, it then distills this information into a CFG. However, producing a CFG from PT is not a trivial task because the traces can contain holes, abrupt interrupts from hardware events, and other low level artifacts. Not only do these need to be removed, but additional metadata also needs to be recovered that will aid the execution unit partitioning described in Subsection 3.2.

Algorithm 1 shows the steps PUMM uses to overcome these challenges. It starts with a linear sequence of instructions I, and generates a list of vertices V, and edge tuples E, representing the trace. The created CFG is based on binary code blocks, which we define as a linear sequence of instructions ending with a branch, indirect call, indirect jump, or return instruction. These blocks are single-entrance, single-exit sequences that may overlap in memory.[4]

The algorithm works by iterating over each traced instruction. At line 10, it checks whether it is currently inside a block

[4]It is also possible to implement this analysis using traditional basic blocks, but doing so incurs a performance cost because newly discovered backward edges may split prior blocks, requiring refactoring of the CFG.

Algorithm 1: Turning a linear instruction trace I into a CFG with vertices V and edge tuples E.

```
1  b,s,c,p,n,V,E ← ∅
2  foreach i ∈ I do
3      if PTStop() then
4          if s then
               // PT turned off for system call
5              s ← False
6          else
               // Unexpected stop, do not create edge
7              p ← ∅
8          end
9      end
10     if b = ∅ then
           // start of new code block
11         b ← i.address
12     end
13     if IsExit(i) then
           // i is last instruction in current block
14         n ← (b, i.address + i.size, c)
15         V ← V ∪ n
16         E ← E ∪ (p, n)
17         p ← n
18         b ← ∅
19         s ← IsSyscall(i)
20         if IsCall(i) then
               // push caller onto context stack
21             c ← c ∪ i.address
22         end
23         if IsRet(i) then
               // pop last caller from context stack
24             c ← c − c.last
25         end
26     else
           // boring instruction
27         s ← False
28     end
29 end
```

and if not, it records the address of the current instruction as the start of a new block. Line 13 checks whether the current instruction is an exit, as previously defined. If it is, then this instruction marks the end of the current block. Lines 14–16 create a new node n, add it to the list of vertices V, and create an edge from the previous node in E. Notice that V and E can contain duplicate entries if the same transfer occurs multiple times in the trace. We describe how they are merged in Subsection 3.2.

When there is no prior block to n, no edge is created in the CFG. This happens for the first block at the start of the trace and for some cases when PT turns off, as handled by lines 3–9. When PT stops due to a system call, which occurs because it is configured to only trace user space, Algorithm 1 keeps the resulting edge. Otherwise, the stop is the result of an interrupt, which could be an exception or context switch. Since PUMM cannot be certain that execution will resume from the interrupted context, Algorithm 1 acts conservatively and does not create an edge, relying instead on the redundancy across subsequent traces to fill holes.

In addition to the start and end address, each block is also labeled with a caller context list c, which tracks all the callers leading up to the current block. This is essentially a shadow stack managed by lines 20–25 that will become relevant in Subsection 3.2.

3.2 Execution Unit Partitioning

The key to PUMM's partitioning is that programs that can process multiple tasks (network requests, input files, etc.) are typically implemented with a main task loop that iterates once per task until an exit condition or signal occurs [26, 31, 33, 39, 40]. Local variables are initialized at the start of a task and the current task does not depend on data from prior ones.

The primary challenge for PUMM is that the main task loop can itself contain loops connected by branching paths. Accidentally identifying one of the inner loops as its own execution unit (under-approximation) can result in premature releasing of quarantined memory, whereas over-approximating the unit incurs additional unnecessary memory overhead.

To solve this challenge, PUMM first post-processes the CFG from Subsection 3.1 to create context sensitive nodes and to insert fake returns for calls across binary objects (e.g., libraries). The former yields a more accurate analysis by accounting for wrappers around library APIs[5] and the latter enables PUMM to analyze the program per-object without having to traverse all the inter-object paths.

Next, PUMM identifies simple circuits (i.e., loops) using the Hawick circuit enumeration algorithm [27], which has a worst case time complexity of:

$$O[(V+E)(C+1)] \tag{1}$$

Where V is the number of vertices, E is the number of edges, and C is the number of circuits. With the circuits located, PUMM starts merging them together to form execution units based on graph dominance. Specifically, given circuits A and B, B is merged with A if the head of A (i.e., its entry node) dominates the head of B. This process is iterated until no more units can be merged.

3.3 Policy Generation & Enforcement

With the execution units identified, PUMM can generate a security policy for the target program. The goal for this step is to identify points in the program where quarantined memory can be safely released to become eligible for reallocation. The main challenge PUMM has to overcome is that the policy must be enforceable at runtime without requiring instrumentation or PT. This is because the system running the target program may not have PT hardware and without source code, instrumentation would have to be inserted dynamically, which incurs a high performance overhead.

Our solution stems from the observation that most execution units begin with initializing variables, including ones that are dynamically allocated. Code blocks calling the underlying memory management library to allocate new variables can be identified by the last return address pushed onto the stack without requiring stack unwinding.[6] Since these calls occur at the beginning of new unit iterations, quarantined memory from prior iterations can be released with low risk, which we evaluate empirically in Section 4.

With this idea in mind, PUMM starts by identifying every caller within a unit to any of the memory allocation functions wrapped by PUMM. For each caller, PUMM checks whether it can be the first caller in a unit iteration, i.e., whether the caller is reachable from the unit head without going through another caller. If it is, then the caller is added to the policy as a release point for the quarantine list.

At runtime, the OS dynamically loads PUMM's memory management wrapper, which in our current prototype covers all the POSIX memory management functions (e.g., `malloc`, `free`). This in turn loads the policy, which is then enforced at any call to the protected functions. Requests that require a buffer to be freed queue the freed address space in the quarantine list, making it ineligible for reallocation. At each call, the caller is identified based on the return address that is pushed onto the stack and if it is in the policy's safe callers list, then the quarantine list is released, making those address spaces eligible for reallocation.

An interesting discovery we make with this design is because frees are being deferred by the quarantine list, it is possible to automatically resolve some double free bugs without having to abort. In PUMM, if both frees are placed in quarantine, they can simply be merged into one release, resolving the violation. However, some system administrators may want double frees to abort regardless, which PUMM can be configured to do. We explain the trade-offs in Section 5.

4 Evaluation

We aim to answer the following questions in our evaluation:

1. *Is PUMM able to prevent UAF exploitation?* We test PUMM against 40 real-world vulnerabilities in 26 popular programs. PUMM prevents all cases from being exploitable with reasonable offline analysis requirements.
2. *How does PUMM's protection compare to prior work?* We evaluate PUMM alongside MarkUs and FFmalloc, the latest systems based on scanning and OTA. PUMM prevents the same vulnerabilities as the other systems while aborting in fewer cases.
3. *What is the overhead of using PUMM?* We run PUMM, MarkUs, and FFmalloc on test workloads for the 26 real-world programs, as well as the SPEC CPU2006

[5] Example: Programs wrap libc's `free` for portability across OSes.

[6] See GNU's `__builtin_extract_return_addr` for details.

standard benchmark. PUMM outperforms MarkUs and FFmalloc, yielding 2.74% less runtime overhead and 52.0% less memory overhead.

4. *How robust is PUMM?* Using 3,000 synthetically injected UAF bugs, we measure the change in memory overhead, number of release points, and bugs that elude PUMM's protection as a function of the number of analyzed profiling traces. PUMM only fails to prevent 4 out of 3,000 bugs from being exploitable. Analyzing more traces results in more release points being added to the policy, yielding a minor improvement to memory overhead with no observable degradation to security.
5. *Is PUMM identifying the main program task loop?* We manually examine the source code of each target program to identify how many tasks it processed during profiling and verify that this number matches the number of units executed at runtime, according to PUMM.

Environment & Baseline Defenses All of our experiments are conducted in a Debian Buster environment using a desktop containing an Intel Core i7-6700K CPU, 16 GB of RAM, and SSD storage. PUMM's CFGs are generated with a caller context sensitivity level of 1, which is typical for binary analysis [59]. To represent the current best designs for scanning and OTA defense, we use MarkUs [1] and FFmalloc [61], respectively. Our prototype's offline analysis is implemented in 600 Python source lines of code (SLoC) on top of the `xed` disassembler and `libipt` Intel PT decoder. Our runtime enforcement is implemented in 450 C SLoC and covers `glibc` and `jemalloc`'s memory management APIs.

Target Programs & Exploits Our evaluation dataset combines 3 datasets of real-world programs and exploits, as well as 1 standard performance benchmark. Specifically, we use all the real-world exploits and programs provided by UAFBench, FFmalloc's original evaluation [61], and our own corpus of programs and exploits found using NIST NVD and Exploit Database. We compile the programs into stripped production binaries using the provided build scripts. For configurable programs like HTTP servers, we use the default settings provided in the code repositories. The performance benchmark we use is SPEC CPU2006, which we pick deliberately to make our results directly comparable to previously reported metrics [1, 61].

Synthetic Bugs Given the challenges in reproducing real-world UAF exploits, we also develop a framework for synthetically injecting UAF bugs into real-world programs to facilitate a larger empirical evaluation of PUMM's robustness, the results of which are presented in Subsection 4.3. This framework is included in our open source repository to ensure experimental reproducibility for future work. At a high level, our framework synthesizes bugs using an external debugger to free a random chunk of allocated memory at a random point in the program's execution, thereby causing a UAF to arise in subsequent memory accesses. The randomness is controlled by a seed value to ensure trials are repeatable. Using this framework, we generate 3,000 synthetic UAF bugs (100 per binary) that when triggered without any defenses, result in control flow redirection, demonstrating their exploitability.

Profiling & Testing Workloads Recall that PUMM relies on offline dynamic profiling to generate policies, which requires tracing runtime program executions. To generate a corpus of inputs for profiling and testing, we start with developer provided test inputs and then fuzz each program for 24 hours using AFL, a popular grey-box fuzzer. This yields a corpus where each input reaches at least 1 novel path. 80% of the inputs are placed in a profiling set and the remaining 20% are withheld for testing. Before performing the split, we remove any inputs that cause a crash to ensure PUMM cannot directly observe a UAF bug during profiling.

Methodology & Calculations We evaluate the security of PUMM and prior defenses using real-world programs and exploits (Subsection 4.1). Since our dataset provides ground truth for each targeted vulnerability, we use a debugger to manually verify the success of each defense at preventing exploitation. For completeness, we distinguish between successful prevention that results in aborted execution versus not aborting. We also record metrics for PUMM's profiling phase to determine feasibility of deployment.

For our performance evaluation, we use both the real-world programs and SPEC, which provides its own *train* and *test* workloads (Subsection 4.2). Overhead is calculated as $(P - B)/B$, where P is the evaluated performance and B is the baseline performance. All time measurements are based on wall-clock time.

In addition to testing PUMM's security on *real-world* exploits, we also evaluate its robustness to code coverage by measuring memory overhead, number of release points, and number of prevented *synthetic* UAF bugs as a function of the number of analyzed traces (Subsection 4.3).

Lastly, we perform a manual verification of the correctness of the execution units identified by PUMM using source code (Subsection 4.4). Specifically, we identify how many tasks are performed during each test input and then compare that to the number of unit iterations. Our intuition is that if PUMM correctly identifies units that encompass all the code required to handle one task, the number of observed unit iterations should equal the number of performed tasks.

4.1 Real-World UAF Prevention

Table 2 presents the results of our security and deployment feasibility evaluations. In each case, we report the vulnerability tested (identified by its CVE, EDB, or bug tracker

Vulnerability	Program	#T	Size (MB)	Mean (MB)	Instructions	CFG Nodes	CFG Edges	Units	Safe	Time (s)	P?
UAFBench											
CVE-2016-3189	bzip2recover	129	3,481	27.0	2,076,381,561	20,621	21,283	206	3	11	Y
CVE-2016-4487	cxxfilt	5,322	147,456	27.7	36,162,246,341	10,269	15,968	3	3	1	Y
CVE-2017-10686	nasm	8,229	51,200	6.2	8,174,108,309	214,211	218,311	1,200	6	5,668	Y
CVE-2018-10685	lrzip	61	888	14.6	161,565,280	54,655	56,930	58	8	3	Y
CVE-2018-11496	lrzip	462	8,192	17.7	1,430,925,696	234,293	243,756	525	9	1,001	Y
CVE-2018-11416	jpegoptim	46	855	18.6	243,499,939	32,294	34,661	252	1	35	Y
CVE-2018-20623	readelf	322	14,336	44.5	1,872,808,345	460,511	477,313	335	6	1,025	Y
CVE-2019-20633	patch	442	906	2.0	222,122,766	100,028	104,405	106	4	45	Y
CVE-2019-6455	rec2csv	564	13,312	23.6	2,756,534,599	501,826	524,818	150	22	23,589	Y
Issue 74	giflib	314	23,552	75.0	5,413,904,821	77,923	82,422	504	10	376	Y
Issue 122	gifsicle	119	19,456	163.5	6,136,138,072	166,219	173,187	334	5	1,711	Y
Issue 73	mjs	1,097	6,041	5.5	1,008,943,580	371,325	377,142	215	8	47,269	Y
Issue 78	mjs	1,152	6,041	5.2	1,032,998,992	366,029	372,651	204	7	48,839	Y
Issue 91	yasm	3,698	19,456	5.2	4,164,443,865	151,547	155,618	2,213	3	32,300	Y
FFmalloc											
CVE-2015-2787	PHP	3,234	26,624	8.2	3,522,225,089	1,004,203	1,008,003	700	33	208,159	Y
CVE-2015-6835	PHP	164	18,432	112.4	3,360,974,313	1,005,633	1,008,766	744	31	222,542	Y
CVE-2016-5773	PHP	191	22,528	117.9	3,721,893,098	1,807,781	1,813,930	1,544	16	277,522	Y
Issue 3515	mruby	593	32,768	55.3	8,635,634,411	660,064	664,548	1,540	1	67,792	Y
CVE-2015-3205	libmimedir	254	2,048	8.1	690,040,157	111,180	111,781	248	7	139	Y
Issue 24613	Python	139	66,560	478.8	14,456,506,475	36,005	57,271	19	46	2	Y
CVE-2019-0568	ChakraCore	487	29,696	61.0	5,312,199,808	1,122,185	1,128,009	3,135	12	208,232	Y
CVE-2020-24346	Nginx	1,997	75,776	38.0	16,905,445,435	668,613	677,158	3,402	3	152,841	Y
NIST NVD & Exploit Database											
CVE-2017-9182	AutoTrace	16	6,963	435.2	1,009,914,907	184,392	192,648	311	25	1,744	Y
CVE-2017-9190	AutoTrace	*Same Program Binary*									Y
CVE-2019-19005	AutoTrace										Y
CVE-2017-11139	GraphicsMagick	506	17,408	34.4	5,290,610,880	256,881	264,072	1,029	1	10,926	Y
CVE-2017-11403	GraphicsMagick	*Same Program Binary*									Y
CVE-2017-12936	GraphicsMagick										Y
CVE-2017-14103	GraphicsMagick										Y
CVE-2017-15238	GraphicsMagick										Y
CVE-2017-18220	GraphicsMagick										Y
CVE-2017-12858	libzip	432	1,331	3.1	433,687,147	100,206	102,717	676	10	639	Y
CVE-2019-17582	libzip	*Same Program Binary*									Y
CVE-2019-6706	Lua	935	16,384	17.5	2,993,395,562	345,947	350,930	2,029	1	43,608	Y
CVE-2015-3890	OpenLiteSpeed	1,352	7,605	5.6	24,255,777,904	24,915	33,636	53	19	1,511	Y
CVE-2010-2939	OpenSSL	566	157,696	278.6	33,461,648,321	14,750	21,898	5	1	1	Y
CVE-2015-8727	Wireshark	240	7,782	32.4	2,316,470,289	58,524	113,571	317	1	91	Y
EDB-39503	Wireshark	*Same Program Binary*									Y
EDB-39529	Wireshark										Y
Issue 144	Gravity	1,422	11,264	7.9	2,170,687,678	484,493	487,791	756	52	34,585	Y
	Average:	1,150	27,201	71.0	6,646,457,788	354,917	363,173	706	11.8	46,407	

Table 2: PUMM evaluation results on the real-world programs and exploits.

ID), the vulnerable program, number of traces collected (#T), their total size, the average size per trace, the total number of instructions recorded, the size of the CFG in terms of nodes and edges, the number of units identified, the number of safe callers located, the total offline analysis time, and whether the vulnerability was prevented. Notice that in some cases we use the same program binary to evaluate multiple vulnerabilities.

In all 40 attacks, PUMM successfully prevents the vulnerability from being exploitable. On average, we profile 1,150 traces per analyzed program, depending on the number of novel inputs uncovered by AFL. Traces average 71 MB in size, which is reasonable. About 6,600,000,000 instructions are recorded per program, on average, with an upper bound of about 36,200,000,000. The resulting CFGs contain about 360,000 nodes and edges, on average, with an upper bound of 1,129,000. We also observe variance in the number of execution units, ranging from 3 to about 3,400 (context sensitive), roughly correlated with the size of the CFG.

The number of safe callers identified in the CFGs vary between programs, with several having only 1. On average, 12 safe callers are identified per program.

On average, it takes PUMM about 12 hours to analyze a program, with PHP being the slowest. This is largely due to the number of novel inputs generated by AFL, requiring PUMM to decode over 1,000 traces per program. Our results in Subsection 4.3 suggest that most of these traces were not necessary to analyze. On average, PUMM is processing 1 trace every 40 seconds. There is no strong correlation between program size and analysis time.

Table 3 compares PUMM's security to prior work, with half circles denoting prevention via aborting or crashing. All three systems either abort or fully prevent all the tested exploits, verifying their effectiveness. We notice that FFmalloc aborts in significantly more tests than the other two systems. Upon investigation, we discover that this is due to a property not described in previous work. Specifically, we discover that defenses that use a quarantine list are able to remediate some instances of UAF by merging conflicting frees together. The simplest example of this occurs in double free bugs, where freeing an address that is already quarantined results in no

Vulnerability	Program	PUMM	MarkUs	FFmalloc
	UAFBench			
CVE-2016-3189	bzip2recover	●	◐	●
CVE-2016-4487	cxxfilt	◐	◐	◐
CVE-2017-10686	nasm	◐	◐	◐
CVE-2018-10685	lrzip	●	●	●
CVE-2018-11496	lrzip	●	●	●
CVE-2018-11416	jpegoptim	●	●	●
CVE-2018-20623	readelf	●	●	●
CVE-2019-20633	patch	◐	●	◐
CVE-2019-6455	rec2csv	●	●	◐
Issue 74	giflib	●	◐	◐
Issue 122	gifsicle	●	●	●
Issue 73	mjs	●	●	●
Issue 78	mjs	●	●	●
Issue 91	yasm	●	●	◐
	FFmalloc			
CVE-2015-2787	PHP	●	●	●
CVE-2015-6835	PHP	●	●	●
CVE-2016-5773	PHP	●	●	●
Issue 3515	mruby	◐	◐	◐
CVE-2015-3205	libmimedir	●	●	●
Issue 24613	Python	●	◐	◐
CVE-2019-0568	ChakraCore	◐	◐	◐
CVE-2020-24346	Nginx	●	●	●
	NIST NVD & Exploit Database			
CVE-2017-9182	AutoTrace	◐	◐	◐
CVE-2017-9190	AutoTrace	●	●	◐
CVE-2019-19005	AutoTrace	◐	◐	◐
CVE-2017-11139	GraphicsMagick	●	●	●
CVE-2017-11403	GraphicsMagick	◐	◐	◐
CVE-2017-12936	GraphicsMagick	●	●	●
CVE-2017-14103	GraphicsMagick	◐	◐	◐
CVE-2017-15238	GraphicsMagick	●	●	●
CVE-2017-18220	GraphicsMagick	●	◐	◐
CVE-2017-12858	libzip	●	●	◐
CVE-2019-17582	libzip	●	●	◐
CVE-2019-6706	Lua	●	●	◐
CVE-2015-3890	OpenLiteSpeed	●	●	●
CVE-2010-2939	OpenSSL	●	●	●
CVE-2015-8727	Wireshark	●	●	●
EDB-39503	Wireshark	●	●	●
EDB-39529	Wireshark	●	●	●
Issue 144	Gravity	●	◐	◐

○: Failure ◐: DoS ●: Full Prevention

Table 3: UAF prevention for PUMM vs. prior work.

change to the program state, allowing execution to continue unhindered. This is the outcome produced by PUMM and MarkUs, whereas FFmalloc does not use a quarantine list and aborts. Notice that while PUMM *can* resolve some double frees, it can also be configured to abort regardless. We elaborate on the trade-offs in Section 5.

4.2 Performance Comparison

Real-World Programs Figures 4 and 5 show the runtime and memory overheads, respectively, for the evaluated systems on the real-world dataset. These metrics are presented in log scale. Tested programs include a wide range of types, such as image processing tools, web servers, network analyzers, cryptography suites, and interpreters. Workload execution times range from a few seconds per input (libzip) to over 30 minutes (OpenLiteSpeed). Maximum memory usage ranges from about 2 MB to over 100 MB. The software versions range from releases from 2010 to 2021.

On average, PUMM outperforms MarkUs and FFmalloc with a runtime overhead of 2.04% and memory overhead of 16.5%. This is 2.74% and 52.0% better than the best combined results of the other two systems. MarkUs performs second best in terms of memory overhead, whereas FFmalloc is second best in terms of runtime.

FFmalloc's memory overhead is over 823% on average and upon investigation, we discover a limitation in FFmalloc's design, which is that in order to reduce the number of system calls being made to allocate memory, which would further increase runtime overhead, FFmalloc requests memory in larger chunks than typical memory managers — at least 4 MB per request. FFmalloc also does not release memory until at least 8 consecutive chunks can be freed, further increasing memory overhead. Since this dataset contains some programs with relatively small memory footprints, these large allocations inflate FFmalloc's memory overhead.

For workloads with shorter execution times, we observe an inflation in runtime overhead reflective of the upfront startup cost of initializing each system. Thanks to PUMM's offline analysis, the policy it enforces at runtime is very efficient to initialize and enforce, giving it an advantage over the other systems. On the other hand, MarkUs and FFmalloc require more time to initialize because they have to create worker threads for scanning or specialized memory pools for OTA. This difference is pronounced in `lrzip`, where we observe a spike in MarkUs' runtime overhead. We also observe that shorter workloads inflate the memory overhead for MarkUs because it cannot release a freed memory chunk until it has verified with a scan that no possible pointers remain. Conversely, we observe that the longer workloads lower the overheads for all three systems, with PUMM still performing best.

One anomaly we observe in the data is the memory overhead for MarkUs on Nginx. In this case, the workload requires thousands of requests to be handled, which takes over 30 minutes in our environment, so it cannot be explained with the previous observations. Upon investigation, we realize that false positive pointer detections are causing MarkUs to quarantine memory longer than necessary. A similar observation was made by the authors of FFmalloc, which inspired them to craft an exploit to exhaust memory [61].

SPEC CPU2006 Figures 6 and 7 show the runtime and memory overheads, respectively, for the evaluated systems on the SPEC benchmark. These metrics are presented in log scale. Once again, PUMM outperforms MarkUs and FFmalloc with an average runtime overhead of 3.12% and memory overhead of 1.87%. By comparison, the overheads for MarkUs are 37.41% and 159.17%, respectively, and 17.09% and 288.19% for FFmalloc. Our results for MarkUs and FFmalloc are within 1% of the numbers reported in their original publications, giving us confidence in the results.

All three systems incur higher overheads compared to the real-world dataset, which is reflective of SPEC's workloads being more CPU and memory intensive. Similar to the real-

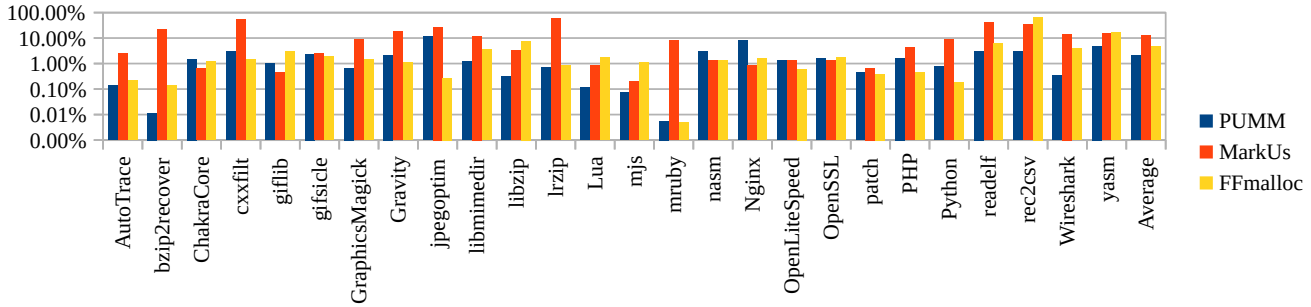

Figure 4: Runtime overhead on the real-world program dataset. PUMM, MarkUs, and FFmalloc's average overheads are 2.04%, 13.02%, and 4.78%, respectively.

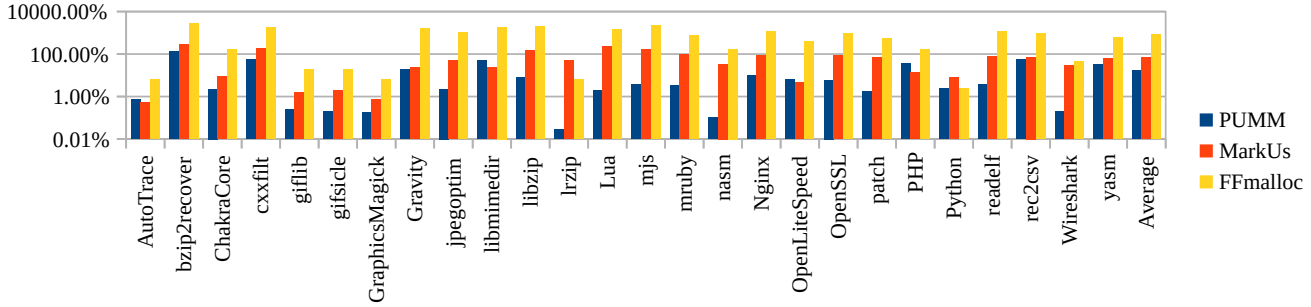

Figure 5: Memory overhead on the real-world program dataset. PUMM, MarkUs, and FFmalloc's average overheads are 16.48%, 68.45%, and 823.90%, respectively.

world programs, MarkUs performs second best in terms of memory overhead whereas FFmalloc performs second best in terms of runtime overhead. We observe that PUMM's performance remains consistent across the SPEC programs with only a minor uptick in runtime overhead for `gamess`. The other systems also experienced increased overhead on this program, with the worst case being MarkUs at over 250%. MarkUs and FFmalloc also experienced increased runtime overheads on `soplex` and `calculix`.

Similar to execution, PUMM's memory overhead is stable across all the SPEC programs, whereas we observe several upticks for the other systems. The worst case for MarkUs is `calculix`, with a memory overhead of over 2,000%. Upon investigation, we believe the root cause of this anomaly is tied to false positives in MarkUs' scanning routine. Unfortunately, the authors of MarkUs did not include this program in their published evaluation, so we cannot verify whether they also encountered this issue. FFmalloc's worst memory overhead is over 1,500%, observed in `libquantum`. This result does not match the overhead published by FFmalloc's authors, despite the other results being consistent. Repeating the experiment yields the same result. Even if we discard these outlier cases, PUMM still significantly outperforms the other systems.

4.3 Code Coverage

Recall from Section 3 that over-approximating an execution unit can increase memory overhead whereas under-approximating can reduce security. With this in mind, we aim to quantify the robustness of PUMM by measuring the change in memory usage, number of release points, and prevented UAFs as a function of the number of analyzed traces. Since our real-world dataset only offers 1 to 6 exploits per program, we turn to synthetically injected bugs to achieve a large scale evaluation, using the technique described at the beginning of this section. Specifically, we evaluate 100 synthetic bugs per program binary, 3,000 in total (26 unique programs, 30 tested versions). For brevity, we will focus on the results for `patch` and Chakra. Results for the rest of the real-world programs are available in the Appendix.

Figure 8 shows the results for `patch`. Our first observation is that as PUMM analyzes more traces, more release points are added to the policy. This is generally true for the whole dataset. Interestingly, at 2 points in PUMM's analysis, new data causes a release point to be dropped from the policy. In short, a newly revealed path makes a release point no longer safe. Despite this, the results show that only 1 of the 100 evaluated bugs is exploitable, and this bug remains exploitable regardless of how many traces are analyzed.

Upon investigation, we discover an edge case not addressed in PUMM's implementation, which we illustrate in Figure 9. This figure contains a portion of `patch`'s CFG, with some simplifications made for readability. The unit head is marked in blue and its lone safe caller is marked in yellow. First, during startup, `patch` allocates and initializes its global data

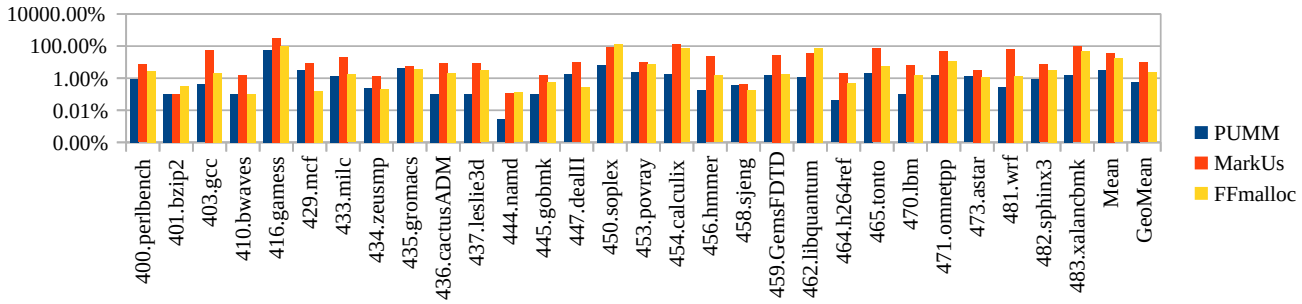

Figure 6: Runtime overhead for the SPEC CPU2006 benchmark. PUMM, MarkUs, and FFmalloc's average overheads are 3.12%, 37.41%, and 17.09% with geometric means of 0.57%, 9.62%, and 2.27%, respectively.

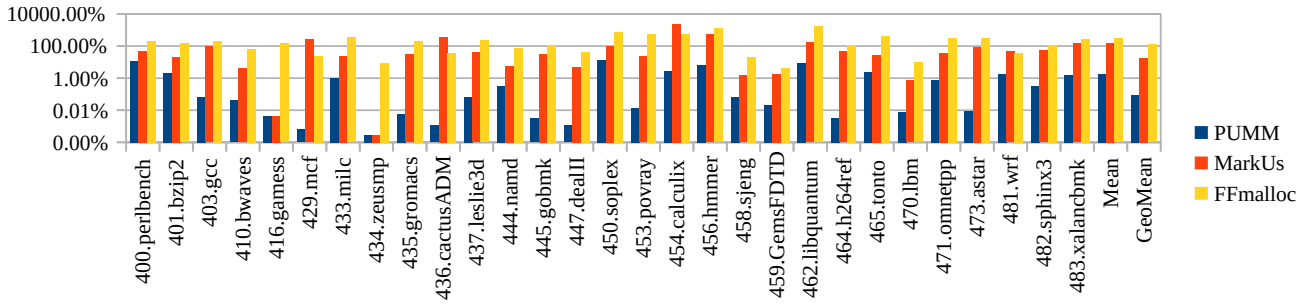

Figure 7: Memory overhead for the SPEC CPU2006 benchmark. PUMM, MarkUs, and FFmalloc's average overheads are 1.87%, 159.17%, and 288.19% with geometric means of 0.09%, 17.76%, and 125.57%, respectively.

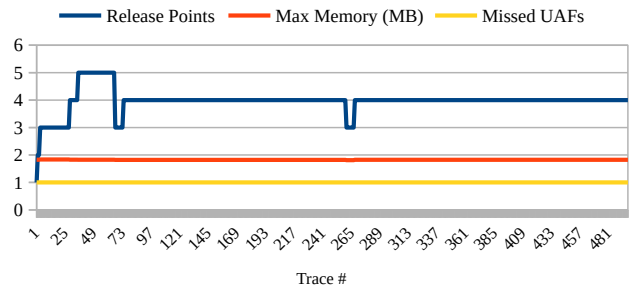

Figure 8: Robustness of PUMM on `patch`.

structures. During this phase, our framework frees one of the structures to synthetically inject a bug, marked in red. This free is quarantined by PUMM, after which the program begins its first iteration of the execution unit, which processes the first input. When the yellow node is reached, PUMM releases the memory that was quarantined during the program's startup, allowing it to be reallocated in subsequent iterations of the unit. Once all the inputs have been processed, `patch` enters its cleanup phase where it then triggers a UAF.

What makes the example in Figure 9 interesting is that the object is defined at the very beginning of `patch`'s execution, but then is not used until `patch`'s cleanup routine at the very end. In other words, this object's data dependencies completely circumvent the execution unit, as denoted by the green arrow. This also explains why the synthetic bug remains exploitable no matter how many traces PUMM analyzes. In total, we find 4 occurrences of this behavior across the 3,000 synthetic bugs we evaluated. These are the only 4 bugs PUMM fails to protect against. This limitation can be addressed by redesigning PUMM to have multiple quarantine lists with different release points, including one for global variables. We leave this direction to future work.

Figure 10 shows the changes in Chakra. This example clearly demonstrates how the memory usage decreases as the number of release points increase. Conversely, the number of prevented UAFs does not change, revealing no signs of under-approximation of the execution units. In summary, we see minor improvements to memory usage across our dataset and *no degradation to security* as more traces are analyzed.

4.4 Manual Verification

Table 4 presents our manual verification of the units identified by PUMM during profiling. Specifically, for each real-world program, we first map the head of the first unit identified by PUMM back to its location in the program's source code, shown as a filename and line number. In parallel, we also study the source code to identify the key data structure that tracks the program's current task.[7] From this information, we identify how many tasks are contained in our test workloads

[7] Prior EUP work has demonstrated how tracking changes to this object at runtime can yield accurate partitions [39].

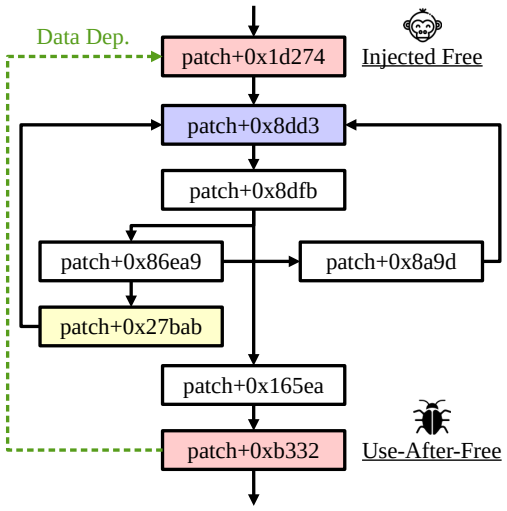

Figure 9: Unprotected synthetic UAF in `patch`. Red denotes the synthetically injected free and subsequent UAF. Blue marks the unit's head and yellow marks the safe caller. The green arrow shows the freed object's first data dependency.

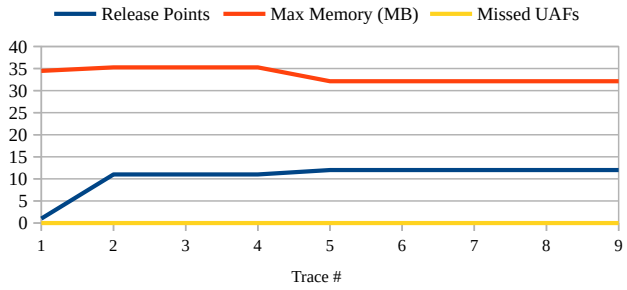

Figure 10: Robustness of PUMM on Chakra.

and how many times the units defined by PUMM are executed.

Notice that for all 26 real-world programs, the number of processed tasks and number of executed units are equal. The counts vary from 2 to 107, mostly based on the size of the workloads. This evidence supports that PUMM is identifying the main task handling loop of the real-world programs, enabling it to accurately dissect the program's execution by task to carefully defer memory reallocations.

5 Limitations

Code Coverage No system based on dynamic profiling can guarantee complete code coverage during profiling, meaning some paths may not be analyzed. However, since PUMM identifies the heads of execution units, which are typically the outermost loop of the program, we expect PUMM to be fairly robust. This claim is supported by our evaluation results. Coverage can be expanded using techniques like fuzzing [45] (as we did) or with symbolic analysis [62].

Program	Unit Head	Task Object	Tasks	Instances
AutoTrace	input-tga.c:278	fp	6	6
bzip2recover	bzip2recover.c:367	bsIn	4	4
ChakraCore	ch.cpp:1270	argInfo	5	5
cxxfilt	cp-demangle.c:2726	di	3	3
giflib	dgif_lib.c:1079	GifFileIn	9	9
gifsicle	gifsicle.c:1530	input_name	9	9
GraphicsMagick	command.c:17396	image_info	8	8
Gravity	gravity.c:222	vm	5	5
jpegoptim	jpegoptim.c:494	infile	15	15
libmimedir	test.c:76	mde	5	5
libzip	ziptool.c:1134	za	107	107
lrzip	runzip.c:314	ss	3	3
Lua	lua.c:445	L	2	2
mjs	mjs.c:12046	mjs	20	20
mruby	gc.c:1711	mrb	5	5
nasm	preproc.c:906	line	20	20
Nginx	njs_shell.c:286	fp	20	20
OpenLiteSpeed	lshttpdmain.cpp:930	m_pServer	8	8
OpenSSL	openssl.c	prog	2	2
patch	main.c:199	patchname	5	5
PHP	php_cli.c:924	script_file	5	5
Python	import.c:163	scan	6	6
readelf	readelf.c:6717	filedata	15	15
rec2csv	rec2csv.c:298	db	3	3
Wireshark	tshark.c:2116	pc	2	2
yasm	yasm.c:500	dir	10	10

Table 4: Manual verification that PUMM's units execute once per processed task.

Security Guarantee Although EUP is largely trusted by the data provenance community, it is ultimately driven by heuristics that can be violated by developers. One such pattern appears in our evaluation of 3,000 synthetic UAFs, which can be resolved by using a separate quarantine list for global variables. Overall, while PUMM demonstrates significantly lower overhead than defenses based on scanning and OTA, it also cannot guarantee security to the degree MarkUs (assuming no pointer obfuscation) and FFmalloc can. However, PUMM still demonstrates valuable protection by halting all 40 real-world vulnerabilities, including all the ones from FFmalloc's original dataset. Similarly, out of 3,000 synthetically generated UAF bugs, PUMM only fails to protect 4, all of which share the same pattern, as described above.

Dynamically Generated Code Since PUMM relies on offline profiling to generate its policies, dynamically generated code cannot release the quarantine list. In the common case where dynamic code is untrusted and isolated by a sandbox (e.g., browsers executing website JavaScript), PUMM identifies all dynamic code as belonging to the same unit as the sandbox. Conversely, if the *entire* program is dynamically generated code, PUMM yields an OTA policy.

Resolving Double Frees As mentioned in Section 3, if two frees quarantine the same memory, PUMM can merge them together to automatically resolve what would otherwise become a double free abort. However, some system administrators prefer for double frees to always abort because otherwise such bugs may go unnoticed and unpatched. For this reason, PUMM provides a configuration setting to preserve the

aborting behavior, if desired.

Compiler Optimized Code One limitation of using stack return pointers for caller identification is that it cannot detect callers that have been optimized by the compiler with tail call elimination. This is sometimes applied when a function ends with a call to another function, which would normally result in two returns being executed sequentially. With the optimization applied, the tail call is replaced with a jump so only one return executes. Consequently, if the eliminated tail call is also a safe caller for PUMM, it will not be eligible for releasing the quarantine list. However, as we show in our evaluation that uses a dataset of optimized production binaries with eliminated tail calls, there has no observable impact to PUMM's performance.

6 Related Work

Execution Unit Partitioning Researchers have explored the use of data provenance to facilitate forensics [5, 24, 29, 31, 33, 35, 37–41, 60], auditing [3, 15–17, 64], alert triage [24, 25], and intrusion detection [4, 13, 47]. The false dependency problem was then formally identified [58], leading to a line of work on EUP [26, 31, 33, 39, 40]. PUMM extends this line of work by exploring whether similar concepts can be applied to UAF prevention.

Secure Allocators Early work in secure allocators [36, 48] demonstrated the possibility of preventing memory safety violations by carefully controlling object placement. For example, UAF can be prevented with high probability by randomizing freed page reuse. Unfortunately, the memory overhead of these systems made them impractical for real-world deployment.

Later designs like DieHard [6] and its successor, DieHarder [46], demonstrated more efficient ways to simulate spatial memory safety. FreeGuard [55] further advanced this work by integrating optimizations from prevalent performance focused allocators. Unfortunately, while FreeGuard achieved better security than the default Linux allocator at a similar performance, it did not match the security of DieHarder. Guarder [56] followed, proposing ways to help bridge the gap between FreeGuard's performance and DieHarder's security. PUMM pursues a different angle, proposing EUP as a fundamental basis for preventing UAF rather than relying on bounded reuse probabilities. There is also work that proposes UAF prevention by restricting reuse to within objects of the same type [2], however this only provides partial UAF protection.

In this work, we directly compare against FFmalloc [61] and MarkUs [1], which offer UAF protection at better performance than the FreeGuard line of work. FFmalloc is the latest design based on OTA, surpassing the performance of Oscar [20] and similar prior work [21]. Likewise, MarkUs is the latest in preventing UAF by retrofitting garbage collection into legacy software and achieves better performance than its prior work [34, 43, 53], several of which also require source code. PUMM is similar to MarkUs in how it temporarily quarantines addresses to prevent reallocation, but distinguishes itself by the use of EUP rather than scanning, which is not limited by pointer hiding [9] in target programs.

Pointer Invalidation Whereas secure allocators prevent UAF by controlling the placement of objects and reuse of pages, it is also possible to focus on the invalidation or nullification of dangling pointers. DangNull [32] tracks all pointers and allocated objects to explicitly nullify pointers once their pointed to object is freed. FreeSentry [63] uses a similar approach, but only flips the top bit of the pointer value, which makes debugging easier. Unfortunately, determining the points-to relationship between objects and pointers is difficult and inaccurate without source code, making these approaches poorly suited for protecting legacy programs. It is also possible to label pointers [12, 42], however this also requires source code and performing the label checks introduces high overhead. This approach also requires tainting to account for pointer arithmetic, which further increases overhead. Alternatively, pointers can be converted into indirect table references [49], enabling invalidation by nullifying a single entry, but this still incurs high overhead.

Use-After-Free Detection It is possible to detect UAF using heavy instrumentation to track memory allocations and accesses, which is implemented in systems like Valgrind [44] and AddressSanitizer [50]. Unfortunately, such approaches have high overhead, making them poorly suited for use outside of testing, and because they are designed for debugging as opposed to security, advanced attackers can circumvent them [32]. Alternatively, it is also possible to fuzz test for UAF bugs [45], however achieving complete code coverage is difficult, which allows some instances to remain undetected.

7 Conclusion

We propose a new design for preventing UAF exploitation based on deferred reallocation and EUP. Using dynamic profiling and analysis, our design identifies autonomous execution units in programs without source code, deferring freed memory from reallocation until after the current unit instance has ended. Our Linux prototype, PUMM, successfully prevents 40 UAF vulnerabilities from being exploited in 26 real-world programs while incurring 2.74% less execution overhead and 52.0% less memory overhead than scanning and OTA-based defenses. Against 3,000 synthetically generated UAFs, only 4 remain exploitable under PUMM's protection.

Acknowledgments

We thank the anonymous reviewers for their helpful and informative feedback. This material was supported in part by the Office of Naval Research (ONR) under grants N00014-19-1-2179, N00014-17-1-2895, N00014-15-1-2162, and N00014-18-1-2662; the Defense Advanced Research Projects Agency (DARPA) under contracts HR00112090031 and N66001-21-C-4024; and Cisco Systems under an unrestricted gift. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of ONR, DARPA, or Cisco.

References

[1] Sam Ainsworth and Timothy M Jones. Markus: Drop-in use-after-free prevention for low-level languages. In *IEEE Symposium on Security and Privacy*, pages 578–591. IEEE, IEEE, 2020.

[2] Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security Symposium*, pages 177–192. Washington DC, 2010.

[3] Adam Bates, Kevin Butler, Andreas Haeberlen, Micah Sherr, and Wenchao Zhou. Let SDN Be Your Eyes: Secure Forensics in Data Center Networks. In *NDSS Workshop on Security of Emerging Networking Technologies*, SENT'14, February 2014.

[4] Adam Bates, Kevin R. B. Butler, and Thomas Moyer. Take Only What You Need: Leveraging Mandatory Access Control Policy to Reduce Provenance Storage Costs. In *USENIX Conference on Theory and Practice of Provenance*, TaPP'15, July 2015.

[5] Adam Bates, Wajih Ul Hassan, Kevin R.B. Butler, Alin Dobra, Bradley Reaves, Patrick Cable, Thomas Moyer, and Nabil Schear. Transparent Web Service Auditing via Network Provenance Functions. In *26th World Wide Web Conference*, WWW'17, 2017.

[6] Emery D Berger and Benjamin G Zorn. Diehard: Probabilistic memory safety for unsafe languages. *ACM SIGPLAN Notices*, 41(6):158–168, 2006.

[7] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking Blind. In *35th IEEE Symposium on Security and Privacy*, 2014.

[8] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-Oriented Programming: A New Class of Code-reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011.

[9] Hans-J Boehm and David Chase. A proposal for garbage-collector-safe c compilation. *Journal of C Language Translation*, 4(2), 1992.

[10] Erik Bosman and Herbert Bos. Framing Signals - A Return to Portable Shellcode. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[11] Fraser Brown, Deian Stefan, and Dawson Engler. Sys: A static/symbolic tool for finding good bugs in good (browser) code. In *29th USENIX Security Symposium*, pages 199–216. USENIX Association, 2020.

[12] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *International Symposium on Software Testing and Analysis*, pages 133–143, 2012.

[13] Frank Capobianco, Christian Skalka, and Trent Jaeger. Accessprov: Tracking the provenance of access control decisions. In *9th USENIX Workshop on the Theory and Practice of Provenance*, 2017.

[14] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-Oriented Programming Without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[15] Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. One primitive to diagnose them all: Architectural support for internet diagnostics. In *12th European Conference on Computer Systems*, EuroSys '17, pages 374–388, New York, NY, USA, 2017. ACM.

[16] Ang Chen, Yang Wu, Andreas Haeberlen, WenchaoZhou, and Boon Thau Loo. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *ACM SIGCOMM Conference*, SIGCOMM'16, pages 115–128, New York, NY, USA, 2016. ACM.

[17] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Differential provenance: Better network diagnostics with reference events. In *14th ACM Workshop on Hot Topics in Networks*, November 2015.

[18] Dustin Childs. Pwn2own 2020 - day one results. https://tinyurl.com/4wrytk4z.

[19] Lucian Constantin. All major browsers fall during second day at pwn2own hacking contest. https://tinyurl.com/uhz4mcxs.

[20] Thurston HY Dang, Petros Maniatis, and David Wagner. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In *USENIX Security Symposium*, pages 815–832, 2017.

[21] Dinakar Dhurjati and Vikram Adve. Efficiently detecting all dangling pointer uses in production servers. In *International Conference on Dependable Systems and Networks*, pages 269–280. IEEE, 2006.

[22] DynamoRIO. Dynamorio. https://dynamorio.org/.

[23] Abdul-Aziz Hariri and Mat Powell. Cve-2020-9715: Exploiting a use-after-free in adobe reader. https://tinyurl.com/fh3j4hd9.

[24] Wajih Ul Hassan, Adam Bates, and Daniel Marino. Tactical Provenance Analysis for Endpoint Detection and Response Systems. In *41st IEEE Symposium on Security and Privacy*, Oakland'20, May 2020.

[25] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. Nodoze: Combatting threat alert fatigue with automated provenance triage. In *Network and Distributed Systems Security Symposium*, 2019.

[26] Wajih Ul Hassan, Mohammad Ali Noureddine, Pubali Datta, and Adam Bates. Omegalog: High-fidelity attack investigation via transparent multi-layer log analysis. In *Network and Distributed System Security Symposium*, 2020.

[27] Kenneth A Hawick and Heath A James. Enumerating circuits and loops in graphs with self-arcs and multiple-arcs. In *FCS*, pages 14–20, 2008.

[28] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE Symposium on Security and Privacy*, pages 969–986. IEEE, 2016.

[29] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS'17, page 377–390, New York, NY, USA, 2017. ACM.

[30] Albert Kwon, Udit Dhawan, Jonathan M Smith, Thomas F Knight Jr, and Andre DeHon. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 721–732, 2013.

[31] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela F Ciocarlie, et al. Mci: Modeling-based causality inference in audit logging for attack investigation. In *Network and Distributed System Security Symposium*, 2018.

[32] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.

[33] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *Network and Distributed Systems Security Symposium*, 2013.

[34] Daiping Liu, Mingwei Zhang, and Haining Wang. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1635–1648, 2018.

[35] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a Timely Causality Analysis for Enterprise Security. In *Network and Distributed System Security Symposium*, NDSS'18, San Diego, CA, USA, February 2018.

[36] Vitaliy B Lvin, Gene Novark, Emery D Berger, and Benjamin G Zorn. Archipelago: Trading address space for reliability and security. *ACM SIGARCH Computer Architecture News*, 36(1):115–124, 2008.

[37] Shiqing Ma, Kyu Hyung Lee, Chung Hwan Kim, Junghwan Rhee, Xiangyu Zhang, and Dongyan Xu. Accurate, low cost and instrumentation-free security audit logging for windows. In *31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 401–410. ACM, 2015.

[38] Shiqing Ma, Juan Zhai, Yonghwi Kwon, Kyu Hyung Lee, Xiangyu Zhang, Gabriela Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. Kernel-supported cost-effective audit logging for causality tracking. In *USENIX Annual Technical Conference*, pages 241–254. USENIX Association, 2018.

[39] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. Mpi: Multiple perspective attack investigation with semantic aware execution partitioning. In *26th USENIX Security Symposium*. USENIX Association, 2017.

[40] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *Network and Distributed Systems Security Symposium*, 2016.

[41] S. Momeni Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan. Holmes: Real-time apt detection through correlation of suspicious information flows. In *IEEE Symposium on Security and Privacy*, Los Alamitos, CA, USA, May 2019. IEEE.

[42] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: Compiler enforced temporal safety for c. In *International Symposium on Memory Management*, pages 31–40, 2010.

[43] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3):477–526, 2005.

[44] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, 42(6):89–100, 2007.

[45] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. Binary-level directed fuzzing for use-after-free vulnerabilities. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses*, pages 47–62, 2020.

[46] Gene Novark and Emery D Berger. Dieharder: Securing the heap. In *17th ACM Conference on Computer and Communications Security*, pages 573–584, 2010.

[47] Jaehong Park, Dang Nguyen, and R. Sandhu. A Provenance-Based Access Control Model. In *10th Annual International Conference on Privacy, Security and Trust*, pages 137–144, 2012.

[48] Bruce Perens. Eletric Fence. https://linux.softpedia.com/get/Programming/Debuggers/Electric-Fence-3305.shtml.

[49] Weizhong Qiang, Weifeng Li, Hai Jin, and Jayachander Surbiryala. Mpchecker: Use-after-free vulnerabilities protection based on multi-level pointers. *IEEE Access*, 7:45961–45977, 2019.

[50] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[51] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[52] Shellphish. how2heap: A repository for learning various heap exploitation techniques. https://github.com/shellphish/how2heap.

[53] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. Crcount: Pointer invalidation with reference counting to mitigate use-after-free in legacy c/c++. In *NDSS*, 2019.

[54] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy*, pages 138–157. IEEE, 2016.

[55] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. Freeguard: A faster secure heap allocator. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 2389–2403, 2017.

[56] Sam Silvestro, Hongyu Liu, Tianyi Liu, Zhiqiang Lin, and Tongping Liu. Guarder: A tunable secure allocator. In *USENIX Security Symposium*, pages 117–133, 2018.

[57] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, 2013.

[58] Dawood Tariq, Maisem Ali, and Ashish Gehani. Towards automated collection of application-level data provenance. *4th USENIX Conference on Theory and Practice of Provenance*, 12:16–16, 2012.

[59] Fish Wang and Yan Shoshitaishvili. Angr - the next generation of binary analysis. In *2017 IEEE Cybersecurity Development*, pages 8–9. IEEE, IEEE, 2017.

[60] Qi Wang, Wajih Ul Hassan, Ding Li, Kangkook Jee, Xiao Yu, Kexuan Zou, Jungwhan Rhee, Zhengzhang Zhen, Wei Cheng, Carl A. Gunter, and Haifeng chen. You Are What You Do: Hunting Stealthy Malware via Data Provenance Analysis. In *Network and Distributed System Security Symposium*, NDSS'20, February 2020.

[61] Brian Wickman, Hong Hu, Insu Yun Daehee Jang, Jung-Won Lim Sanidhya Kashyap, and Taesoo Kim. Preventing use-after-free attacks with fast forward allocation. In *30th USENIX Security Symposium*. USENIX Association, 2021.

[62] Carter Yagemann, Matthew Pruett, Simon Chung, Kennon Bittick, Brendan Saltaformaggio, and Wenke Lee. Arcus: Symbolic root cause analysis of exploits in production systems. In *USENIX Security Symposium*, 2021.

[63] Yves Younan. Freesentry: Protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS*, 2015.

[64] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. Secure Network Provenance. In *ACM Symposium on Operating Systems Principles*, 2011.

(a) AutoTrace. (b) `bzip2recover`. (c) `cxxfilt`. (d) `giflib`.

(e) `gifsicle`. (f) GraphicsMagick. (g) Gravity. (h) `jpegoptim`.

(i) `libmimedir`. (j) `libzip`. (k) `lrzip`. (l) Lua.

(m) `mjs`. (n) `mruby`. (o) `nasm`. (p) Nginx.

(q) OpenLiteSpeed. (r) OpenSSL. (s) PHP. (t) Python.

(u) ReadELF. (v) `rec2csv`. (w) Wireshark. (x) `yasm`.

Figure 11: Additional results for the PUMM robustness experiment. For programs where multiple versions were tested, only one graph is presented for brevity. We observe no major differences between different versions of the same program.