HeapTherapy+: Efficient Handling of (Almost) All Heap Vulnerabilities Using Targeted Calling-Context Encoding

Qiang Zeng[†], Golam Kayas[‡], Emil Mohammed[‡], Lannan Luo[†], Xiaojiang Du[‡], and Junghwan Rhee[§]
[†]*University of South Carolina* [‡]*Temple University* [§]*NEC Lab*
{zeng1,lluo}@cse.sc.edu, {golamkayas,tuf58189,dux}@temple.edu, {rhee}@nec-labs.com

***Abstract*—Exploitation of heap vulnerabilities has been on the rise, leading to many devastating attacks. Conventional heap patch generation is a lengthy procedure requiring intensive manual efforts. Worse, fresh patches tend to harm system dependability, hence deterring users from deploying them. We propose a heap patching system HEAPTHERAPY+ that simultaneously has the following prominent advantages: (1) *generating patches without manual efforts*; (2) *installing patches without altering the code* (so called *code-less patching*); (3) *handling various heap vulnerability types*; (4) *imposing a very low overhead*; and (5) *no dependency on specific heap allocators*. As a separate contribution, we propose *targeted calling context encoding*, which is a suite of algorithms for optimizing calling context encoding, an important technique with applications in many areas. The system properly combines heavyweight offline attack analysis with lightweight online defense generation, and provides a new countermeasure against heap attacks. The evaluation shows that the system is effective and efficient.**

***Index Terms*—Heap memory safety, automatic patch generation, dynamic analysis, calling context encoding.**

I. Introduction

As many effective measures for protecting call stacks get deployed (such as canaries [1], reordering local variables [2], and Safe SEH [3]), heap vulnerabilities gain growing attention of attackers. Heap vulnerabilities can be exploited by attackers to launch vicious attacks. The recent Heartbleed [4] and WannaCry [5] attacks demonstrate the dangers. For instance, the WannaCry ransomware uses the EternalBlue exploit, which makes use of a heap buffer overwrite vulnerability to hijack the control flow of the victim program [5]. It is notable that heap memory vulnerabilities nowadays are frequently exploited to launch ROP-based attacks [6], which makes heap memory protection an even more urgent and important task.

There are a variety of heap vulnerability types. The following types are among the most commonly exploited types.[1] (1) **Buffer overflow:** it includes both *overwrite* and *overread*. By overwriting a buffer, the attack can manipulate data adjacent to that buffer and launch various control-data or non-control-data attacks, while exploitation of overread can steal sensitive information in memory, such as address space layout and private keys. (2) **Use after free:** it refers to accessing memory after it has been freed. If the memory space being reused is under the control of attackers, use-after-free bugs can be exploited to launch various attacks, such as control flow hijacking. (3) **Uninitialized read:** exploitation of such vulnerabilities can leak sensitive information.

Many approaches have been proposed to tackle heap vulnerabilities. A large body of research focuses on detecting, preventing or mitigating heap attacks (and other memory-based attacks) [8]–[25]. They usually incur a large overhead or/and can only handle a specific type of heap vulnerabilities. For example, MemorySanitizer [20] is a dynamic tool that detects *uninitialized read*; however, it incurs an average of 2.5x of slowdown and 2x of memory overhead. AddressSanitizer [8], which detects *overflows* and *use after free* online, is deemed *fast*, but still incurs 73% slowdown and 3.4x memory overhead. As another example, HeapTherapy [19] proposes an efficient *heap buffer overflow* detection and response system; however, it does *not* detect and handle *uninitialized read* and *use after free*.

When examining the spectrum of heap security measures, we notice that handling heap vulnerabilities through ***patching*** has been much less studied. Patching, however, is an indispensable step for handling vulnerabilities in practice. Over decades, conventional patch generation and deployment have suffered serious limitations. First, the patch generation is a lengthy procedure. Even for security sensitive bugs, it takes those *big* vendors 153 days on average from vulnerability report to patch availability [26]. A study finds that only 65% of vulnerabilities in software running on a typical Windows host have patches available at vulnerability disclosure [27]. This provides opportunities for attackers to exploit the unpatched vulnerabilities on a large scale [28]. For resource-constrained small software companies, it takes even longer time.

Second, given a vulnerability, its fresh patches may have not been thoroughly tested, and thus tend to introduce stability issues and even logic errors. Although waiting for mature patches can reduce the risk, it makes the exploitation window longer. This has been a dilemma in patch deployment [29].

We propose a heap patching system that does not have the limitations above. Our insight is that, by changing the configuration of heap memory allocation, *all* the aforementioned heap vulnerabilities can be addressed without altering the program code and, hence, no new bugs are introduced. According to the configuration information, the allocator enhances its handling

[1]*Double free was* frequently exploited; but many popular allocators, such as the default allocator in glibc [7], have built-in double free detection now.

(i.e., allocation, initialization and deallocation) of buffers that are vulnerable to attacks, called *vulnerable buffers* and, more importantly, applies security enhancement only to them (rather than all buffers) to minimize the overhead. We refer to it as *Heap Patches as Configuration*.

We accordingly build our system HEAPTHERAPY+. Unlike *HeapTherapy* [19], which generates defenses for overflow bugs by combining online in-memory trace collection and post-mortem core dump analysis, HEAPTHERAPY+ consists of a *heavyweight* offline patch generation phase and a *lightweight* online defense generation phase. In the offline patch generation phase, we use *shadow memory* to scrutinize attacks and achieve bit precision level. We group buffers according to their allocation-time calling contexts. Buffers that share the same calling context as the buffer exploited by the attack are regarded as vulnerable buffers. The calling context along with other information is collected to generate *patches*, i.e., the configuration information. Next, in the online defense generation phase, the configuration information is loaded and the stored calling context information guides the allocator to recognize vulnerable buffers. It properly combines detailed and powerful offline analysis and highly efficient online defenses.

However, if call stack walking (as used by `gdb`) is used for obtaining calling contexts, it can incur significant slowdown, especially for allocation-intensive programs [30]–[33]. We thus use *calling context encoding*, which *continuously* represents the current calling context in one or a few integers [30]. By reading the integer(s), the encoded calling context, called *Calling Context ID (CCID)*, can be obtained. By comparing the CCID for the current buffer allocation with the CCIDs stored in the patches, the online system can swiftly determine whether the new buffer is vulnerable.

Moreover, we propose *targeted calling context encoding*, which is a suite of algorithms that can optimize many well-known calling encoding methods, such as PCC [30], PCCE [31], and DeltaPath [32]. Since calling context encoding is an important technique with many applications, the optimization algorithms constitute a separate contribution.

Installing a heap patch does not change the program code. Specifically, a heap patch is in the form of a $\langle key, value\rangle$ tuple, where the *key* is the allocation-time CCID of the vulnerable buffer and the *value* indicates the vulnerability type and the parameter(s) for applying the online defense. The patches are read into a hash table upon program initialization. It thus takes only O(1) time to determine whether a new buffer is vulnerable. The online defense is enforced by intercepting heap buffer allocation and deallocation. Both the hash table initialization and the buffer allocation/deallocation interception are implemented in a shared library, and are *transparent* to the underlying heap allocator. We thus do *not* need to modify the heap allocator or depend on a specific allocator.

None of the techniques used in HEAPTHERAPY+, except for *targeted calling context encoding*, is new. However, static analysis, code instrumentation, offline attack analysis, and online defense generation are creatively combined to build a new countermeasure against heap attacks. A comprehensive evaluation is performed, showing that HEAPTHERAPY+ is effective and efficient. We make the following contributions.

- We properly combine heavyweight offline attack analysis and lightweight online defense generation to build a new heap defense system that simultaneously demonstrates the following good properties: (1) *patch generation without manual efforts*, (2) *code-less patching*, (3) *versatile* handling of heap buffer overwrite, overread, use after free, and uninitialized read, (4) *imposing a very small overhead*, and (5) *no dependency on specific allocators*.
- We propose *targeted calling context encoding*, a suite of algorithms that can optimize calling context encoding, and demonstrate its application to our system.

II. RELATED WORKS

Given the large body of research on heap memory safety, we do not intend to make an exhaustive list of work on the problem. Instead, we compare HEAPTHERAPY+ with other automatic patch generation techniques, and then examine critical techniques used in our system.

A. Automatic Patch/Defense Generation

With attack inputs in hand, generating patches/defenses automatically has been a highly desired goal. We divide previous researches towards this goal into the following categories.

Bytes pattern based signature generation. Given *a large number* of attack inputs, many systems (such as Honeycomb [34], Autograph [35], and Polygraph [36]) generate signatures by extracting common bytes patterns from the inputs. However, such methods usually need *many* attack samples in order to correctly mine patterns, and cannot work when only one or very few attack inputs are available. False positives may be raised when benign inputs happen to match the signatures. Plus, attackers can mutate the inputs to bypass the detection. In addition, these systems usually have deployment difficulty in handling compressed or encrypted inputs.

Semantics based signature generation. Tools like COVERS [37], Hamsa [38], TaintCheck [39] and the work by Xu et al. [40] propose methods to generate semantics-based signatures; e.g., spotting the target system call ID used upon control flow hijacking and filtering out inputs that contain the ID. They are very effective in handling certain control flow hijacking attacks, but it is unknown how they can be applied to addressing overread and uninitialized read. They also have deployment difficulty in handling compressed and encrypted attack inputs and may incur false positives.

Tracking faulty instructions. By replaying the attacks, some systems try to pinpoint faulty instructions that are exploited by the attacks and try to generate patches to fix them; such systems include VSEF [41], Vigilant [42], PASAN [43] and AutoPag [44]. A frequently employed insight is that a tainted input, e.g., due to overwrite, should not be used to calculate the indirect jump address. It is unknown how such systems can handle attacks beyond control flow hijacking, e.g., buffer overread attacks. Plus, the deployment of the patches requires code update, just like conventional code patching.

Trial and error for patch generation. Some systems propose genetic programming based program generation [45], template based patch generation [46], and patch generation via machine learning [47] to generate many patches, and test each of them against prepared *test cases* until one patch passes all of them. However, it usually takes a lot of effort to prepare well-structured test cases with a decent test coverage. Other systems keep generating candidate patches based on certain criteria until one can recover the program execution [48], [49]. There is no guarantee a patch can be generated using these methods, and the generated patch may introduce logic errors.

While there are many works on automatic defense/patch generation, most of the proposed systems suffer one or more of the following limitations: deployment difficulties, false positives, requiring many attack inputs or test cases. Unlike existing automatic patch generation systems, HEAPTHERAPY+ supports easy deployment without code updates, guarantees zero false positives, requires only *one* attack input, and handles multiple types of heap vulnerabilities.

B. Calling Context Encoding

A *calling context* is the sequence of active function calls on the call stack. It carries critical information about dynamic program behavior. It thus has been widely used in debugging, testing, anomaly detection, event logging, performance optimization, and profiling [32]. For example, logging sensitive system calls is a practice in many systems. Recording the calling context of the system call provides important information about the sequence of program components that gets involved and leads to the call.

Obtaining calling contexts through stack walking is straightforward but very expensive [30]. A few encoding techniques, which represent a calling context using one or very few integers, have been proposed to continuously track calling contexts with a low overhead. The *probabilistic calling context* (PCC) technique [30] computes a probabilistically unique integer ID, essentially a hash value, for each calling context, but does not support decoding. *Precise calling context encoding* (PCCE) [31] stems from *path profiling* [50] and supports decoding. *DeltaPath* [32] improves PCCE by supporting virtual function calls and large-sized programs. A relevant but different problem is path encoding [50], which represents execution paths (within a control flow graph) into integers.

Similar to *targeted calling context encoding*, another work [51] also aims to minimize the encoding overhead but uses a very different idea. It performs offline-profiling runs to establish the mappings between stack offsets and calling contexts. It fails if the calling context of interest does not appear in the profiling runs. Its reported decoding failure rate is as high as 27%. Finally, it does work if variable-size local arrays (allowed in C/C++) are used.

C. Calling Context-Sensitive Defenses

Calling context was applied to various areas *beyond* debugging decades ago. As an example, a region-based heap allocator tags heap objects with allocation-time calling contexts [52]. Recently, calling context is used to generate context sensitive defenses [19], [33], [40], [41], [53]. In particular, *Exterminator* [33] also generates context-sensitive heap defenses. However, our system HEAPTHERAPY+ differs from Exterminator in multiple aspects. (1) Exterminator performs online probabilistic attack detection (e.g., when an overflow occurs, it may or may *not* detect it), while HEAPTHERAPY+ performs offline deterministic attack analysis and patch generation. How to apply patches generated by heavyweight offline analysis to lightweight online defense generation is non-trivial and solved by our work. (2) Exterminator does not handle overread or uninitialized-read, while HEAPTHERAPY+ handles all the frequently exploited heap vulnerability types including overwrite, overread, use after free, and uninitialized read. (3) Exterminator relies on a custom heap allocator that incurs large overheads, while HEAPTHERAPY+ does not; the defense of HEAPTHERAPY+ is transparent to the underlying allocator. (4) Exterminator uses the expensive stack walking to retrieve calling contexts, while *targeted calling context encoding* is proposed and used in HEAPTHERAPY+. But the two works share the insight in calling context-sensitive heap patches, which we do *not* claim as our contribution.

D. Shadow Memory

Our offline heavyweight analysis makes use of shadow memory [54], which tags every byte of memory used by a program with some information. For example, by tagging a memory region as inaccessible, a *read zone* is created. Despite its powerful capabilities in dynamic analysis, it incurs very high overheads. The implementation in Memcheck, which is built in Valgrind, incurs 22.2x slowdown [54]. AddressSanitizer improved it significantly with many functionalities cut, but still incurs 73% speed slowdown [8]. Our system extends shadow memory by associating every heap buffer with its calling context ID.

HEAPTHERAPY+ does not propose new techniques except targeted calling context encoding, but properly combines heavyweight offline analysis (based on shadow memory) and lightweight online defenses (based on allocation/deallocation interception and calling context encoding). It resolves the challenge of applying offline analysis results to online defenses.

III. PROBLEM STATEMENT AND ARCHITECTURE

A. Problem Statement

Similar to conventional patch generation, our system uses collected inputs that reproduce the bug for vulnerability investigation and patch generation. Given a program $\mathbb{P}$ that contains a heap vulnerability $\mathcal{V}$ and an attack input $\mathcal{I}$ that exploits $\mathcal{V}$,[2] our system outputs a patch $\mathcal{P}$, which, once installed, can

[2]Actually, an attack input is not *required*. “*Steps to reproduce*”, which is a regular part of a bug report [55], suffices.

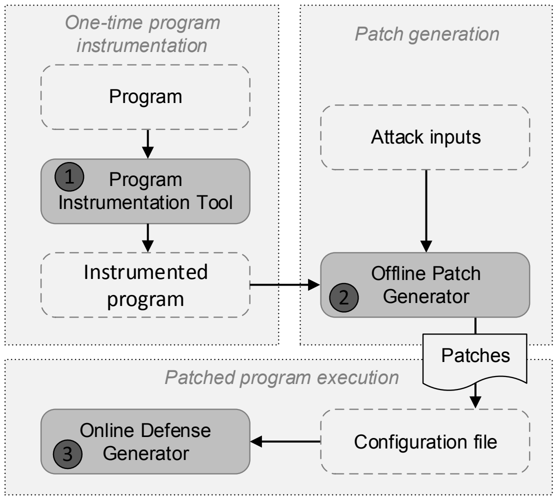

One-time program instrumentation
Program
1 Program Instrumentation Tool
Instrumented program
Patch generation
Attack inputs
2 Offline Patch Generator
Patches
Patched program execution
3 Online Defense Generator
Configuration file

Fig. 1. System architecture.

defeat attacks that exploit $\mathcal{V}$. We consider the *three* frequently exploited heap vulnerability types described in Section I.

But our system differs from conventional patch generation in the following aspects. (1) Instead of relying on lengthy manual investigation, patches can be generated instantly without human intervention. (2) Rather than updating the program $\mathbb{P}$ to fix vulnerabilities, patches are written into a configuration file $\mathcal{C}$ to take effect, without introducing new bugs.

B. System Architecture

As shown in Figure 1, the system consists of the following components: (1) A *Program Instrumentation Tool*: it builds the calling context encoding capability into the program (Section IV). Program instrumentation is an *one-time* effort. Because of the simplicity of the instrumentation, its correctness can be verified automatically. The instrumented program is then used for both offline patch generation and the online system. (2) An *Offline Patch Generator*: it automatically generates the patch by replaying the attack (Section V). (3) An *Online Defense Generator*: it is a *shared library* that (a) loads the patches from the configuration file $\mathcal{C}$, and (b) intercepts buffer allocation operations for recognizing vulnerable buffers and generate defenses online (Section VI).

C. Calling-Context Sensitive Patches

In order to generate a patch $\mathcal{P}$ based on attack analysis, it is critical to extract some *invariant* among attack instances. Such invariant then can be used to design protection against future attacks that also exploit $\mathcal{V}$.

Our observation is that attacks that exploit $\mathcal{V}$ usually share some attack-time calling context (e.g., the sequence of active function calls that lead to a buffer overflow due to a `memcpy` call). If we trace the program execution backward, these vulnerable buffers should share the allocation-time calling context, which we call a *vulnerable calling context* and can be used as an invariant to generate the patch $\mathcal{P}$.

IV. Targeted Calling Context Encoding

Simple call stack walking for retrieving calling contexts would incur a large overhead, especially for programs with intensive heap allocations [30]. There exist several efficient calling context encoding techniques, such as [30]–[32]. We propose *targeted calling context encoding*, which is a suite of algorithms that can be used to optimize these encoding techniques. The insight is that when the *target functions*, whose calling contexts are of interest, are known, many call sites do not need to be instrumented and thus the overhead can be significantly reduced. Specifically, if some functions never appear in the calling contexts that lead to the target functions, they do not need to be instrumented (Section IV-A); moreover, if one function has only one call site that can reach the target function, then its instrumentation can also be pruned (Sections IV-B and IV-C).

While we believe the optimization algorithms can benefit other encoding techniques [31], [32], to make the discussion concrete (and based on our choice of the encoding technique for heap patching), we use *Probabilistic Calling Context* (PCC) [30] to demonstrate the application of the proposed optimizations. According to PCC, at the prologue of each function, the current calling context ID (CCID), which is stored in a thread-local integer variable V, is read into a local variable t; right before each call site, V is updated as $V = 3 * t + c$, where c is a constant integer unique for each call site.[3] This way, V continuously stores the current CCID. Hence, the current CCID can be obtained conveniently by reading V. With PCC, however, it may happen that multiple calling contexts obtain the same encoding due to hash collisions. It is shown practically and theoretically that the chance of hash collision is very low [30]. It is worth noting that a hash collision in our system means that a non-vulnerable buffer may be recognized as a vulnerable buffer and get enhanced. Any of our enhancements do not change the program logic, so *a hash collision can cause unnecessary overhead, but does not affect the correctness of our system.*

We call the original encoding algorithm that take all the call sites into consideration as Full-Call-Site (FCS) instrumentation. The three famous encoding algorithms, PCC [30], PCCE [31] and DeltaPath [32] all enforce FCS. Figure 2(a) shows that all the call sites in those red nodes are instrumented, and T_1 and T_2 are the target functions. The less call sites are instrumented, the smaller overhead is incurred.

A. Targeted-Call-Site (TCS) Optimization

FCS blindly instruments all the call sites in a program. In practice, very often users are only interested in the calling contexts that end at one of a specific set of target functions, such as security-sensitive system calls and critical transaction calls. In our case, we are only interested in calling contexts when the allocation APIs (such as `malloc`, `calloc`, `calloc`, `memalign`, `aligned_alloc`) are invoked. It is

[3]The encoding in PCCE [31] and DeltaPath [32] basically adopts $V = t + c$, where c is calculated according to their encoding algorithms.

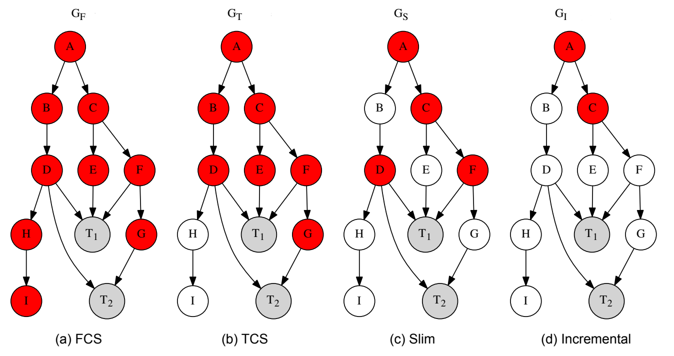


Fig. 2. Comparison of different encoding optimization algorithms using an example call graph. The gray nodes, T_1 and T_2, are target functions; red nodes indicate functions whose call sites are instrumented; and white nodes indicate functions whose call sites are not instrumented. For the simplicity of presentation, the example does not include ***back edges***. Our algorithms can handle back edges without problems as shown in Algorithm 1.

unnecessary to instrument functions that may never appear in the call stacks when these target functions are invoked.

We thus propose the first optimization, *Targeted-Call-Site* (TCS), where only the call sites that may appear in the calling contexts of target functions are instrumented. To conduct the TCS optimization, reachability analysis on the call graph of the program is performed. Given a call graph $G = \langle V, E \rangle$, where V is the set of nodes representing functions of the program and E the set of function calls, and a set of functions $\mathcal{F}$, we perform reachability analysis to find edges that can reach any of the functions in $\mathcal{F}$, and only call sites corresponding to these edges are instrumented.

Figure 2(b) shows the instrumentation result of TCS. As the edges DH and HI cannot reach any of the target functions T_1 and T_2, they are pruned from the instrumentation, reducing the set of call sites that need to be instrumented.

B. Slim Optimization

On the basis of TCS, there is still potential to further prune the set of call sites to be instrumented. In a call graph, a node can be classified as either a *branching* or *non-branching* one: a branching node is one that has multiple outgoing edges that can reach (one of) the target functions. *Our insight* is that the purpose of call site instrumentation is to make sure different calling contexts can obtain different encoding values; given a non-branching node, whether or how its contained call sites are instrumented does not affect the *distinguishability* of the encoding results. Thus, we propose to avoid instrumenting the call sites in those non-branching nodes.

For example, as shown in Figure 2(c), according to the Slim optimization, all call sites in the non-branching nodes, B and E, are excluded from the instrumentation set.

C. Incremental Optimization

The two optimization algorithms treat all target functions as a whole. Our another *insight* is that when the call to a target function is intercepted for analysis or logging purpose, the analyzer or logger usually knows the target function. In our case, when `malloc` and `memalign` are intercepted, different interception functions will be invoked.

Therefore, we can use the pair of $\langle$ `Target_fun, CCID` $\rangle$ (rather than `CCID` alone) to distinguish different calling contexts. Based on this insight, we propose another optimization algorithm that can further reduce the number of instrumented call sites. A node is a *true branching node* if it has two or more outgoing edges that reach the same target function. That is, if a node has multiple outgoing edges, each of which reaches a different target function, it is called a *false branching node*. The idea of the Incremental encoding is to avoid instrumentation the call sites in a false branching node.

In Figure 2, node A is a true branching node, as its two outgoing edges can reach the same node T_1 (and T_2 as well). So is node C, as its two outgoing edges can reach T_1. Thus, only the call sites that correspond to AB, AC, CE, CF need to be instrumented. Take the calling contexts of T_2 as an example, the instrumentation at AB and AC is sufficient to distinguish the two calling contexts that reach T_2.

Algorithm 1 Incremental Optimization.

Input: A call graph $CG = \langle N, E \rangle$, and the set of target functions $T \subseteq N$, where N is the set of functions and E edges.
Output: The edges in E to be instrumented.

```
1:  function FILTER($T, CG = \langle N, E \rangle$):
2:      $InstrumentationSet \leftarrow \{\}$
3:      for $t \in T$ do
4:          $VisitedNodes \leftarrow \{\}$
5:          $Queue.push(t)$
6:          for $n \leftarrow Queue.pop()$ do
7:              $VisitedNodes.push(n)$
8:              for each $e = \langle m, n \rangle$ of the incoming edges of $n$ do
9:                  if $m \notin VisitedNodes$ then
10:                     $Queue.push(m)$
11:         for $n \in VisitedNodes$ do
12:             $ReachingEdgesSet \leftarrow \{\}$
13:             for each $e = \langle n, m \rangle$ of outgoing edges of $n$ do
14:                 if $m \in VisitedNodes$ then
15:                     $ReachingEdgesSet.push(\langle n, m \rangle)$
16:             if $ReachingEdgesSet.size() > 1$ then
17:                 $InstrumentationSet.push(ReachingEdgesSet)$
18:     return $InstrumentationSet$
```

Algorithm 1 shows the algorithm for incremental optimization. Line 3 illustrates the idea of processing each target function incrementally. For each target function, Lines 4–17 are to find true branching nodes relative to it. Specifically, Lines 4–10 are a backward breadth-first search; as it omits nodes already visited (Line 9), it can correctly handle *back edges*. Then Lines 11–17 are to find true branching nodes.

V. Offline Attack Analysis and Patch Generation

The Offline Patch Generator component runs the vulnerable program using the attack input and generates the patch as part of the dynamic analysis report. It is built on dynamic binary instrumentation and shadow memory of Valgrind [54]. As shown in Figure 3, for every bit of the program memory, a *Validity* bit (V-bit) is maintained to indicate whether the accompanying bit has a valid value (i.e., initialized); instructions are inserted for

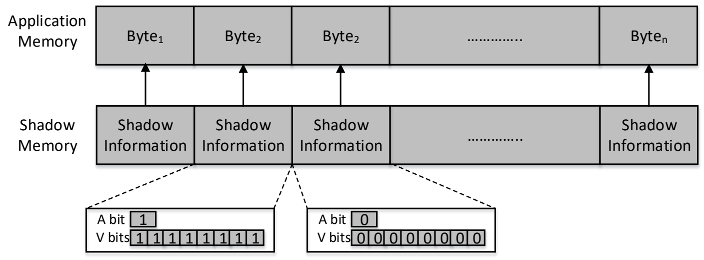
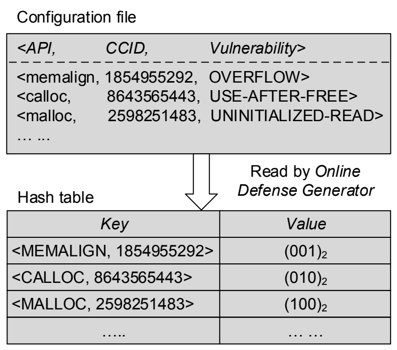


Fig. 3. Shadow memory.



```
<API,          CCID,           Vulnerability>
<memalign, 1854955292,  OVERFLOW>
<calloc,       8643565443,  USE-AFTER-FREE>
<malloc,       2598251483,  UNINITIALIZED-READ>
… …
```

Key	Value
<MEMALIGN, 1854955292>	$(001)_2$
<CALLOC, 8643565443>	$(010)_2$
<MALLOC, 2598251483>	$(100)_2$
…..	… …

Fig. 5. Patches read into a hash table.

```
typedef struct {
  uint32_t i;
  uint8_t c;
} A;
A y, *p = (A *) malloc( sizeof(A) );
p->i = 0; p->c = 'f';
y = *p;
```

Fig. 4. Legal uninitialized read due to padding.

the propagation of V-bits when data copy occurs (e.g., when a word is read from memory to a register); for every byte of the memory location, an *Accessibility* bit (A-bit) is maintained to indicate whether the memory location can be accessed.

When a heap buffer is allocated, the returned memory is marked as accessible but invalid. Each buffer is surrounded by a pair of *red zones* (16 bytes each), which are marked as inaccessible. When a heap buffer is *free*-ed, its memory is set as inaccessible. In addition, whenever a heap buffer is allocated, the current calling context ID (CCID) is recorded and associated with the buffer.

(1) Detecting overflows: A buffer overflow will access the inaccessible red zone appended to the buffer and get detected.

(2) Detecting use after free: A *free*-ed buffer is set as inaccessible and then added to a FIFO queue of freed blocks. Thus, the memory is not immediately made available for reuse. Any attempts to access any of the blocks in the queue can be detected. The maximum total size of the buffers in the queue is set as 2GB by default, which is large enough for the exploits we investigated, and can be customized. In Section IX, we discuss how to handle it if the quota is insufficient.

(3) Detecting uninitialized read: To detect uninitialized read, an attempt is to report any access to uninitialized data, but this will lead to many false positives. For instance, given the code snippet in Figure 4, most of the compilers will round the size of `A` to 8 bytes; so only 5 bytes of the heap buffer is initialized (and the V-bits for the remaining 3 bytes are zero), while the compiler typically generates code to copy all 8 bytes for `y = *p`, which would cause false positives due to accessing the 3 bytes whose V-bits are zero. To avoid false positives due to padding, we check the V-bit of a value only when it is used to decide the control flow (e.g., `jnz`), used as a memory address, or used in a system call (as the kernel behavior is not tracked). As every bit of the program has a V-bit, bit-precision detection of uninitialized read is achieved. Moreover, *origin tracking* is used to track the use of invalid data back to the uninitialized data (such as a heap buffer) when a warning is raised, which allows us to retrieve the allocation-time CCID associated with the vulnerable buffer.

When an attack is detected, the patch is generated in the form of ⟨ `FUN, CCID, T`⟩, where `FUN` is the function used to request the heap buffer (such as `malloc`, `memalign`), `CCID` is an integer representing the allocation-time calling context ID of the vulnerable buffer, and `T` is a three-bit integer representing the vulnerability type (the three bits are used to indicate OVERFLOW, USE-AFTER-FREE, UNINITIALIZED-READ, respectively). Example patches are shown in the upper graph in Figure 5.

How to handle `realloc`: If the new size is smaller than the original size, the cut-off region is marked as inaccessible. If the new size is larger, the added region is set as accessible but invalid. The allocation-time CCID associated with the buffer is also updated with the value upon the `realloc` invocation.

How to handle multiple vulnerabilities: An attack input may exploit multiple vulnerabilities. For example, the Heartbleed attack exploits both uninitialized read and overread. In order to handle the case that an attack exploits multiple vulnerabilities, we resume the program execution upon warnings. Plus, once the V bits for a value have been checked, they are then set to *valid*; this avoids a large number of chained warnings. Finally, a script is used to process the many warnings according to the origin (i.e., the vulnerable buffer) of those warnings and generate patches.

VI. Code-less Patching and Online Defenses

When the patched program is started, as shown in Figure 5, the *Online Defense Generator* library has an initialization function[4] that reads patches from the configuration file and stores them into a *hash table*, where the key of each entry is ⟨ `ALLOCATION_FUNCTION, CCID`⟩ and the value is the vulnerability type(s) and parameters, if any, for generating online defenses. *Note once the hash table is initialized, its memory pages are set as read only.*

The Online Defense Generator library intercepts all heap memory allocation operations. Whenever a heap buffer is allocated, the name of the allocation function along with the

[4]`__attribute__((constructor))` is used to declare the function.

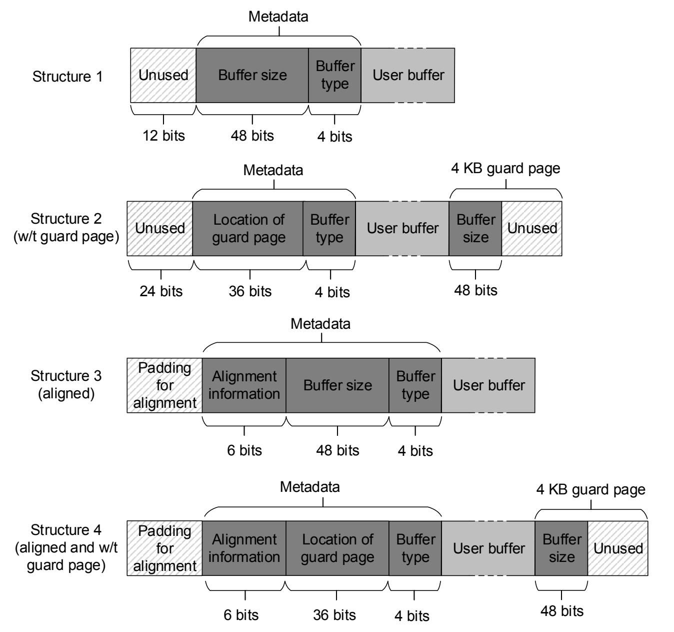


Fig. 6. Buffer structures. Note how we pack the metadata into only one word (64 bits) preceding the *user buffer*.

current CCID is used to search in the patch hash table, which takes only O(1) time. If there is no match, the buffer does not need to be enhanced; otherwise, the buffer is enhanced based on the associated vulnerability type(s) and parameters.

Several considerations make the design of online defenses challenging. (1) In some cases, the same buffer may be vulnerable to multiple attacks, such as uninitialized read and overflow. (2) In addition to handling `malloc` and `free`, the system needs to support a family of other allocation functions, such as `realloc` and `memalign` (aligned allocation). These challenges are well resolved by our system. Another complexity is that we maintain heap metadata ourselves, such as the buffer size (to support `realloc` correctly), vulnerability type(s), the buffer alignment information, and the location of the guard page, so that *our system can work without having to change the underlying allocator or rely on its internals.*

(1) Handling overflows: If the buffer is vulnerable to overflows, a guard page is appended to it to prevent such attacks. While the guard page can effectively prevent overflows, they are known to be prohibitively expensive when being applied to every buffer. In our system, however, the guard page is precisely applied to vulnerable buffers, and the resulting overhead is dramatically reduced.

As shown in Figure 6, Structure 2 is used for non-aligned buffers, while Structure 4 is used for aligned buffers (allocated using `memalign`, etc.). When a heap allocation request is intercepted, the requested size is increased to accommodate the word for *metadata* and the guard page (as well as necessary padding following the user buffer to ensure the guard page is page aligned). The address of the user buffer is returned to service the user program.

TABLE I
A SUMMARY OF THE USE OF BUFFER STRUCTURES.

Vulnerability type	Not aligned	Aligned
Not Vulnerable	Structure 1	Structure 3
Overflow	Structure 2	Structure 4
Use-after-free	Structure 1	Structure 3
Uninitialized read	Structure 1	Structure 3
Overflow & Use-after-free	Structure 2	Structure 4
Overflow & Uninitialized read	Structure 2	Structure 4
Use-after-free & Uninitialized read	Structure 1	Structure 3
Overflow & Use-after-free & Uninitialized read	Structure 2	Structure 4

The *metadata* word contains rich information and is worth detailed interpretation. (1) In all structures, the least significant four bits is called the *buffer type* field, where three bits represent the *vulnerability type* (one bit is used to indicate each of the three vulnerability types, i.e., `Overflow`, `Use after Free`, and `Uninitialized Read`) and one bit indicates whether the buffer is *aligned*. (2) 36 bits are used to indicate the location of the guard page. Currently, 64-bit operations systems only use a 48-bit virtual address space; plus, a guard page is 4KB=$2^{12}B$ aligned. Thus, $48 - 12 = 36$ bits are sufficient. A guard page is set as inaccessible using `mprotect`. The user buffer size information is stored as the first word of the guard page, and it is needed for supporting `realloc`. (3) If the buffer is aligned (Structure 3 and Structure 4), there is a padding field whose size depends on the alignment size. The alignment size information is needed to determine the buffer address given the address of the User Buffer upon a `free` call. As the alignment size is always a power of two (i.e., 2^n), we only need 6 bits to store the value of $n \in [0, 64]$, which then can be used to calculate the alignment size.

(2) Handling use after free: If an allocation is not aligned, the buffer takes Structure 1; otherwise, Structure 3. The metadata word uses 48 bits to store the user buffer size. When a buffer vulnerable to use after free is to be *free*-ed, it is put into an FIFO queue of freed blocks to defer the reuse.

In our system, only buffers vulnerable to use-after-free are put into the queue, such that given the same quota the time a freed buffer stays in the queue is much lengthened, which hence significantly increases the difficulty of exploitation of a use-after-free vulnerability for it increases the uncertainty entropy a freed buffer is reused by attackers.

(3) Handling uninitialized read: Similar to the above, if the allocation is not aligned, the buffer takes Structure 1; otherwise, Structure 3. The *user buffer* region is initialized with zeros before it is returned to the user program.

Table I summarizes how different buffer structures are used for handling different cases, including when multiple vulnerabilities affect the same buffer. If there is a threat of overflow, Structure 2 or Structure 4 is used to accommodate the guard page depending on whether the allocation call is

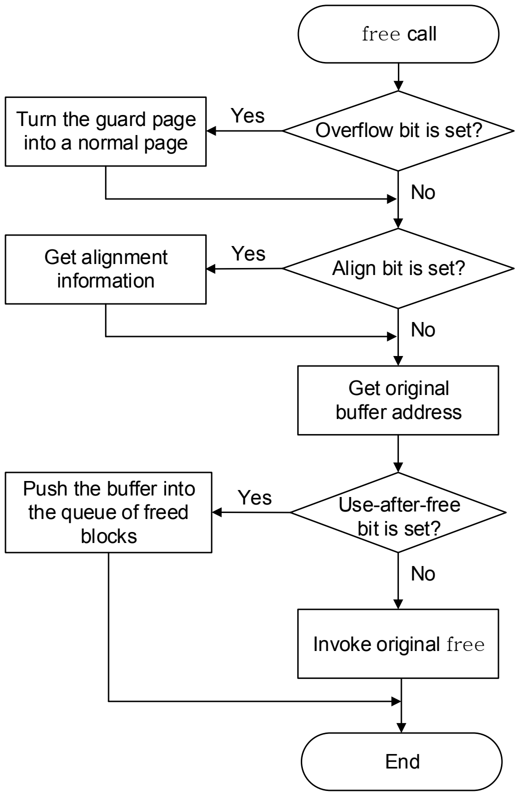

free call
Overflow bit is set?
Yes
Turn the guard page into a normal page
No
Align bit is set?
Yes
Get alignment information
No
Get original buffer address
Use-after-free bit is set?
Yes
Push the buffer into the queue of freed blocks
No
Invoke original free
End

Fig. 7. Handling `free()`.

`memalign`. Whenever there is use after free, upon being freed the buffer is put into the freed-blocks queue to defer the reuse of these buffers.

How to handle `free()` calls? A particular advantage of our system is that it supports the deployment of heap patches without modifying the underlying allocator. It works solely by intercepting the memory allocation calls. On the other hand, it complicates the handling of freeing buffers.

As shown in Figure 7, when `free(p)` is invoked the Online Defense Generator intercepts the call and handles it as follows. (1) If the Overflow bit in the metadata word is set, the location information of the guard page is retrieved and the guard page is set as accessible using `mprotect`. (2) Based on the user buffer address `p`, the initial address of the buffer `pi` is calculated. Specifically, if the buffer was not allocated using `memalign`, `pi = p - sizeof(void*)`; otherwise, the alignment size `A` is retrieved and `pi = p - A`. (3) If the Use-after-Free bit is set, the block is put into the queue of the freed blocks; otherwise, the buffer is released using the original `free` API of the underlying allocator.

VII. Other Implementation Details

Program Instrumentation Tool. We add a pass into LLVM, which performs the call graph analysis to determine the set of call sites to be instrumented and then instruments them. This implementation has the limitation of requiring the program source code to be available. Given the simplicity of the analysis and instrumentation, we suppose a binary-only implementation path (e.g., via *Dyninst* [56]) is viable.

Offline Patch Generator. This component is built on the basis of Valgrind [54]. We reuse its shadow memory functionality and modify the tool to handle allocation and deallocation. Significant effort has been saved by making use of Valgrind, which in the meanwhile is a mature dynamic analysis tool. The implementation over Valgrind thus benefits us to analyze various complex real-world programs successfully.

Online Defense Generator. It is implemented as a shared library, which reads the patches in the configuration file to the hash table. Once the initialization is done, the hash table memory pages are set as read-only. The library also interposes the buffer allocation calls (such as `malloc` and `free`) to enforce the runtime protection. Note that `malloc` and `free` are usually implemented in a shared library, typically `libc`. Thus, as long as our shared library (which also implements `malloc` and `free`) is loaded before `libc`, calls to these functions will be dispatched to our library. In Linux, by specifying our shared library during compilation using `LDLIBS+=` (or loading it using `LD_PRELOAD`), we can ensure it is loaded before `libc`. Our implementation of `malloc` and `free`, in addition to enforcing the protection, invokes libc APIs to perform the real allocation/deallocation. So it does not change the underlying heap allocator or rely on its internals.

TABLE II
VULNERABLE PROGRAMS USED IN THE EVALUATION. UR AND RAF STAND FOR UNINITIALIZED READ AND USE AFTER FREE, RESPECTIVELY.

Program	Vulnerability	Reference
Heartbleed	UR & Overflow	CVE-2014-0160
bc-1.06	Overflow	Bugbench [57]
GhostXPS 9.21	UR	CVE-2017-9740
optipng-0.6.4	UaF	CVE-2015-7801
tiff-4.0.8	Overflow	CVE-2017-9935
wavpack-5.1.0	UaF	CVE-2018-7253
libming-0.4.8	Overflow	CVE-2018-7877
SAMATE Dataset	Variety	23 heap bugs [58]

VIII. Evaluation

We have evaluated HEAPTHERAPY+ in terms of both effectiveness and efficiency. We not only evaluate it on the SPEC CPU2006 benchmarks and many vulnerable programs, but also run the system with real-world service programs. Our experiments use a machine with a 2.8GHZ CPU, 16G RAM running 16.04 Ubuntu and Linux Kernel 4.10.

A. Effectiveness

To evaluate the effectiveness of our system HEAPTHERAPY+, we run it on a series of programs, as shown in Table II, which contain a variety of heap vulnerabilities. We aim to evaluate (1) whether the Offline Patch Generator can correctly determine the vulnerability type and generate patches; and (2) whether the generated patches can effectively prevent attacks from exploiting those heap vulnerabilities.

Heartbleed Attacks. Heartbleed was a notorious vulnerability of `OpenSSL` and affected a large number of services [59]. By sending an ill formed heartbeat request and receiving the response, the attacker can steal data from the vulnerable services, such as private keys and user account information. While Heartbleed is widely known as a heap buffer over-read vulnerability, actually the attacker can exploit two different vulnerabilities: over-read and uninitialized read. Specifically, the vulnerable heap buffer has 34KB, while the size l of the data stealing from the buffer can be up to 64KB. If $l < 34K$, the attack is just an uninitialized read that leaks data previously stored in the buffer; otherwise, it is a mix of uninitialized read and over-read [60].

A service was created using the *OpenSSL* utility `s_server`.[5] We then collected different attack inputs from Internet, and used one of them to generate the patch. Our Offline Patch Generator correctly identified it as a mix of uninitialized read and overflow and output the patch. The patch was then automatically written into the configuration file of the Online Defense Generator, which was able to precisely recognize and enhance the vulnerable buffers. We then tried different attack inputs, and no data was leaked except for the zeros filled in the buffers.

bc-1.06. *bc*, for basic calculator, is an arbitrary-precision calculator language. Some versions of its implementation contain a heap buffer overflow vulnerability. We obtained a buggy version of this program from BugBench, a C/C++ bug benchmark suite [57], and collected a malicious input that overflows buffers and corrupts the adjacent data. By feeding the input into our Offline Patch Generator that ran the buggy program, an overflow patch was generated. With the patch deployed, our system successfully stopped the attack before it corrupted any data.

GhostXPS 9.21. `GhostXPS` is an implementation of the Microsoft XPS document format built on top of `Ghostscript`, which is an interpreter/renderer for PostScript and normalizing PDF files. It is the leading independent interpreter software with the most comprehensive set of page description languages on the market today. Some versions of `GhostXPS` contain an uninitialized read vulnerability that can be exploited using a crafted document. We collected a buggy version of *GhostXPS* from their git repository and the malicious document input. In the offline patch generation phase, the uninitialized read attack was detected and a patch was generated. During the online heap protection phase, the attack was not able to steal any data, except for zeros, from memory.

optipng-0.6.4. `OptiPNG` is a `PNG` image optimizer that compresses image files to a smaller size without losing any information. Specific versions of this optimizer allow the attacker to exploit a use-after-free vulnerability and execute arbitrary code via crafted PNG files. We collected a vulnerable version (`optipng-0.6.4`) and a malicious *PNG* image. The Offline Patch Generator correctly identified the attack and generated a patch. The Online Defense Generator made use of the patch to recognize the vulnerable buffers and defeated the use-after-free attacks by deferring the deallocation of vulnerable buffers.

tiff-4.0.8. `TIFF` provides support for ”Tag Image File Format”, commonly used for sorting image data. In `LibTIFF 4.0.8`, there is a heap buffer overflow in the `t2p_write_pdf` function in `tools/tiff-2pdf.c`. We were able to generate the patch, which could successfully prevent the overflow.

SAMATE Dataset. We evaluated our system on the `SAMATE Dataset`, which is maintained by NIST [58] and contains 23 programs with heap buffer overflow, uninitialized read, or use after free vulnerabilities. Our system successfully generated patches for all of them and prevented the vulnerabilities from being exploited.

B. Efficiency

We compared the overhead incurred by the different calling context encoding algorithms, and measured the overall speed overhead and memory overhead incurred by our system. We used our LLVM-based implementation to measure the efficiency of different calling context encoding algorithms.

1) Overhead Comparison of Different Calling Context Encoding Algorithms: To measure the execution time overhead imposed by different calling context encoding algorithms, we applied them to the programs in the SPEC CPU2006 Integer benchmarks, and measured the execution time when different encoding techniques were applied, normalized using the execution time when no encoding is applied. Compared to FCS (Full Call-Site Instrumentation) proposed in [30], which incurred 2.4% of slowdown for C/C++ programs, the other three encoding algorithms proposed by us, that is, TCS (Targeted Call-Site Instrumentation), Slim, and Incremental, incurred 0.6%, 0.5%, and 0.4% of slowdown, receptively. While the saved execution time itself is small, it gains up to 6x of speed up. We believe the proposed encoding algorithms can have many applications far beyond memory protection; plus, when they are applied to Java programs, where FCS may incur more than 35% of overhead [32], the speed up due to our algorithms could make a significant difference.

As the encoding works by inserting instructions into the programs, we also measured the program size increase. The results are shown in Table III. While FCS increased the binary size by an average of 12% when compared to the uninstrumented binaries, TCS, Slim and Incremental incurred only 6%, 4.5%, and 4.4% of size increase, respectively.

2) Efficiency of HEAPTHERAPY+*:* To evaluate the runtime overhead, we ran our system on both SPEC CPU2006 Integer benchmarks and a set of real-world service programs.

SPEC CPU2006. The speed overhead incurred by

[5]In order to support the interposition of the allocation operations, we compiled OpenSSL using the `OPENSSL_NO_BUF_FREELIST` compilation flag to disable the use of freelists.

TABLE III
SPEC CPU2006 BENCHMARK PROGRAM SIZE INCREASE, IN PERCENTAGE, DUE TO DIFFERENT ENCODING ALGORITHMS.

Benachmark	FCS(%)	TCS(%)	Slim(%)	Incremental(%)
400.perlbench	19.6	16.2	15.9	15.9
401.bzip2	8.8	0.12	0.12	0.12
403.gcc	18.6	14.7	13.6	13.6
429.mcf	0.53	0.53	0.53	0.53
445.gobmk	4.8	3.2	2.5	2.5
456.hmmer	18.9	5.9	2.4	1.2
458.sjeng	10.6	0.08	0.08	0.08
462.libquantum	15	7.7	7.7	7.7
464.h264ref	8.3	3.6	1.8	1.8
471.omnetpp	15.8	7.2	6.7	6.7
473.astar	7.0	7.0	0.2	0.2
483.xalancbmk	14.5	4.1	3.8	3.8

TABLE IV
SPEC CPU2006 BENCHMARK HEAP ALLOCATION STATISTICS.

Benachmark	malloc	calloc	realloc
400.perlbench	346,405,116	0	11,736,402
401.bzip2	174	0	0
403.gcc	23,690,559	4,723,237	44,688
429.mcf	5	3	0
445.gobmk	606,463	0	52,115
456.hmmer	1,983,014	122,564	368,696
458.sjeng	5	0	0
462.libquantum	1	121	58
464.h264ref	7,270	170,518	0
471.omnetpp	267,064,936	0	0
473.astar	4,799,959	0	0
483.xalancbmk	135,155,553	0	0

HEAPTHERAPY+ can be divided into four parts: (1) overhead due to instrumentation, which has been presented above; (2) overhead due to interposition of heap memory allocation calls; (3) overhead due to maintaining the meta data of each buffer (such that our system does not rely on the internal details of the underlying allocator); (4) overhead due to patch deployment, which causes the security measures to be applied to vulnerable buffers.

In order to measure the overhead incurred due to patch deployment, we select a set of allocation-time CCIDs (Calling-Context IDs) as hypothesized vulnerable ones as follows. First, for each benchmark program, we rank all of its allocation-time CCIDs according to their frequencies during the profiling execution (that is, how many heap buffers have been allocated under that calling context). Next, we pick the CCIDs with *median* frequencies as the hypothesized vulnerable ones. Finally, we regard the heap buffers with those allocation-time CCIDs as ones vulnerable to overflows (the other two vulnerability types are much less expensive to treat), and generate corresponding patches for them.

Figure 8 shows the measurement results. The overhead due to interposition is 1.9%, and the overhead for maintaining the buffer metadata (excluding the interposition overhead 1.9% and calling context encoding overhead 0.4%) is 2.0%. Note that the two parts of overhead can be largely eliminated if our system is integrated into the underlying heap allocator. When zero patch is installed, the overhead is 4.3%. When one patch is installed, the overhead increases by only 0.4% (and reaches 4.7%). The total overhead is 5.2% when five patches are installed. One outlier is `400.perlbench`, which has the most intensive heap allocations. Table IV records the heap allocation statistics for each SPEC CPU2006 benchmark.

We also measured the memory consumption of benchmark programs, and used a script that can compute the memory overhead in terms of the average Resident Set Size (RSS) for the benchmark programs. That script reads the `VmRSS` field of `/proc/[pid]/status`. The sampling rate is 30 times per second, and the average of the readings is reported. Figure 9 shows the memory consumption overhead normalized over native program execution, and the average overhead is only 4.3%. The overhead is due to the metadata (e.g., buffer size) our systems maintains for each buffer, and can be largely eliminated if our system is integrated into the underlying heap allocator. Note that guard pages themselves do not increase the use of memory, since they are virtual pages.

Service Programs. We also evaluated our system on two popular service programs: `Nginx` and `MySQL`. We used `Nginx` 1.2, and measured the throughput overhead by sending requests using `Apache Benchmark`. Different numbers of concurrent requests from 20 to 200 were used, and the throughput was compared with that of native execution. The average throughput overhead is only 4.2%. We used `MySQL` 5.5.9 and its built-in script `mysql-stress-test.pl` to measure the throughput overhead. There was no observable throughput overhead. The memory overhead in both cases was negligible. Note that the memory overhead is proportional with the number of *live* buffers.

Comparison with State of the Art. MemorySanitizer [20] detects uninitialized read only at an average of 2.5x slowdown on SPEC CPU2006. AddressSanitizer [8] detects overflows and use-after-free only, and the average slowdown on CPU2006 is 73% and the memory overhead is 3.37x. They make use of shadow memory for online detection, while our work uses it for offline analysis only; they detect attacks on both heaps and call stacks. Exterminator (see Section II-C) incurs only 7.2% slowdown on CPU2006, but it generates the heap buffer overflow defense probabilistically over multiple runs. Unlike Exterminator, HeapTherapy [19] deterministically generates the overflow defense in a single run, and it is the first work that introduces calling context encoding to memory defense generation, but it cannot handle uninitialized read and use-after-free. Cruiser [10] uses a concurrent thread to scan the heap integrity and achieves very high efficiency (5% slowdown on CPU2006), but it can only detect overwrites.

Summary. The evaluation shows that HEAPTHERAPY+ is effective but efficient. It generates defenses for a variety of heap vulnerabilities in a deterministic way. On SPEC CPU2006, the most optimized calling context encoding incurs only 0.4% of slowdown, a 6 times of speed boost compared to PCC [30]. When five patches are installed, the total speed

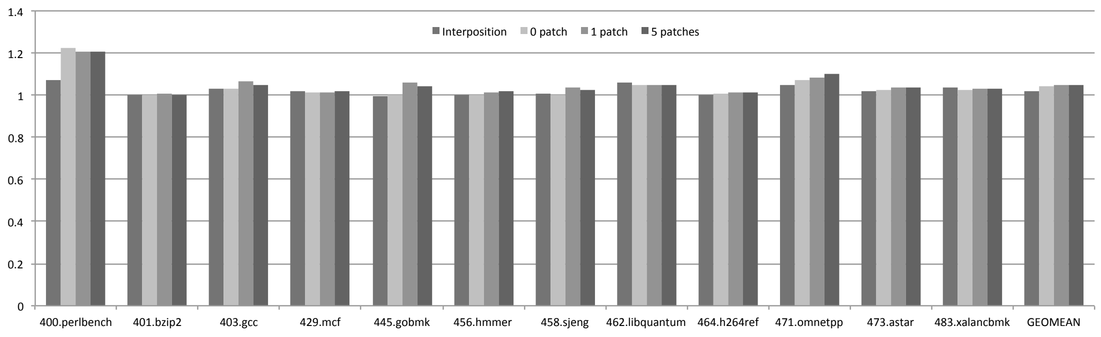


Fig. 8. Normalized execution time overhead imposed to SPEC CPU2006 benchmarks (smaller is better), when only interposition is applied (1.9%), no patch is installed (4.3%), one patch is installed (4.7%), and five patches are installed (5.2%), respectively.

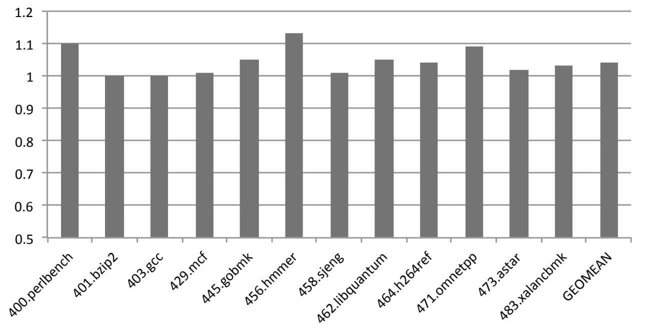


Fig. 9. Normalized memory overhead imposed to SPEC CPU2006 benchmarks when HEAPTHERAPY+ runs. The average memory overhead is 4.3%.

overhead is only 5.2%, most of which is due to interposing allocation/deallocation and maintaining metadata; it means that the overhead can be further reduced if our system is integrated into the underlying heap allocator. The throughput overhead on real-world service programs is very low or negligible.

IX. DISCUSSION

A limitation of HEAPTHERAPY+ is that it can only handle the overflow caused by continuous writes or reads, which is the main form of buffer overflows, though. Overflows due to discrete reads or writes cannot be handled by HEAPTHERAPY+. Plus, if an overflow runs over an array which is an internal field of a structure, HEAPTHERAPY+ cannot detect it. These limitations are shared by many existing countermeasures against buffer overflows, such as AddressSanitizer [8] and HeapTherapy [19]. A common challenge for heap security tools that work via interception of allocation calls is to make them work with custom allocators. Existing works like MemBrush [61] may be leveraged to locate custom memory allocations and address this challenge.

More precisely, the *patches* generated by HEAPTHERAPY+ are *configurable runtime defenses*, since they do not completely fix a bug (e.g., DoS can still be triggered via overflows). Our goal is not to replace conventional patching. Instead, it is to complement the conventional patching procedure by providing immediate protection when patches are not available or fresh patches still need more time for testing.

It may occur that a heap vulnerability can be exploited via multiple CCIDs. Thus, the attacker may develop different attack inputs to exploit buffers with those CCIDs. However, whenever the attack exploits a buffer allocated in a new calling context, our system simply treats it as a new vulnerability and starts another defense generation cycle. Based on our evaluation and previous researches on context-sensitive defenses [19], [33], [40], [48], such cases are rare.

When analyzing the use-after-free attack for programs that have large memory profiles, the memory quota for the FIFO queue of freed blocks may be drained. In this case, we can replay attacks in multiple executions; specifically, we divide the whole space of CCIDs into N subspaces, and each of the N executions defers the deallocation of buffers that have the allocation-time CCIDs in one of the subspaces. Now, each execution is expected to consume $1/N$ of the memory.

X. CONCLUSIONS

We have combined heavyweight offline attack analysis and lightweight online defense generation to build a new heap memory defense system HEAPTHERAPY+. It demonstrates how shadow memory that incurs tens of times of slowdown can be used for generating defenses that imposes a very small overhead. It has many prominent advantages: (1) *patch generation without manual efforts*, (2) *code-less patching*, (3) *versatile* handling of heap buffer overwrite, overread, use after free, and uninitialized read, (4) *imposing a very small overhead*, and (5) *no dependency on specific allocators*. The evaluation shows that is effective and efficient. The speed overhead is only 5.2% when five patches are installed on SPEC CPU2006 benchmarks, and the overhead can be further reduced is the system is integrated into the underlying allocator.

In addition, we have proposed *targeted calling context encoding*, which achieves six times of speed boost compared to the prior encoding technique and may interest researchers applying or building calling context encoding techniques.

ACKNOWLEDGMENT

This project was supported by NSF CNS-1815144 and NSF CNS-1856380. We thank the anonymous reviewers for their invaluable comments and constructive suggestions.

References

[1] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks." in *USENIX Security Symposium*, 1998, pp. 63–78.

[2] PaX, "The PaX project," https://pax.grsecurity.net/.

[3] Microsoft, "Directives Reference," https://msdn.microsoft.com/en-us/library/16aexws6.aspx.

[4] Codenomicon, "The Heartbleed Bug," http://heartbleed.com/.

[5] N. Grossman, "EternalBlue Everything There Is To Know," https://research.checkpoint.com/eternalblue-everything-know/.

[6] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, p. 2, 2012.

[7] Linux, "mallopt-Linux man page," https://linux.die.net/man/3/mallopt.

[8] K. Serebryany, D. Bruening, A. Potapenko, and D. D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *USENIX ATC*, 2012, pp. 309–318.

[9] T. Liu, C. Curtsinger, and E. D. Berger, "Doubletake: fast and precise error detection via evidence-based dynamic analysis," in *38th International Conference on Software Engineering (ICSE)*, 2016, pp. 911–922.

[10] Q. Zeng, D. Wu, and P. Liu, "Cruiser: Concurrent Heap Buffer Overflow Monitoring Using Lock-free Data Structures," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011, pp. 367–377.

[11] D. Tian, Q. Zeng, D. Wu, P. Liu, and C. Hu, "Kruiser: Semi-synchronized Non-blocking Concurrent Kernel Heap Buffer Overflow Monitoring," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, 2012.

[12] E. D. Berger and B. G. Zorn, "DieHard: Probabilistic Memory Safety for Unsafe Languages," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006, pp. 158–168.

[13] G. Novark and E. D. Berger, "DieHarder: Securing the Heap," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, 2010, pp. 573–584.

[14] S. Silvestro, H. Liu, C. Crosser, Z. Lin, and T. Liu, "FreeGuard: A Faster Secure Heap Allocator," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 2389–2403.

[15] P. Akritidis, "Cling: A memory allocator to mitigate dangling pointers." in *USENIX Security Symposium*, 2010, pp. 177–192.

[16] Y. Younan, "FreeSentry: Protecting Against Use-After-Free Vulnerabilities Due to Dangling Pointers," in *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS)*, 2015.

[17] E. van der Kouwe, V. Nigade, and C. Giuffrida, "DangSan: Scalable Use-after-free Detection," in *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, 2017, pp. 405–419.

[18] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, "Preventing Use-after-free with Dangling Pointers Nullification." in *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS)*, 2015.

[19] Q. Zeng, M. Zhao, and P. Liu, "HeapTherapy: An Efficient End-to-End Solution against Heap Buffer Overflows," in *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2015, pp. 485–496.

[20] E. Stepanov and K. Serebryany, "MemorySanitizer: fast detector of uninitialized memory use in C++," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015, pp. 46–55.

[21] K. Lu, C. Song, T. Kim, and W. Lee, "UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 920–932.

[22] A. Milburn, H. Bos, and C. Giuffrida, "SafeInit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities," in *Proceedings of the 24th Network and Distributed System Security Symposium (NDSS)*, 2017.

[23] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007, pp. 89–100.

[24] Q. Zeng, M. Zhao, and P. Liu, "Poster: Targeted therapy for program bugs," in *the 35th IEEE Symposium on Security and Privacy*, 2014.

[25] S. Sidiroglou, G. Giovanidis, and A. D. Keromytis, "A dynamic mechanism for recovering from buffer overflow attacks," in *International Conference on Information Security*, 2005, pp. 1–15.

[26] S. Frei, "The Known Unknowns," https://www.techzoom.net/Papers/The_Known_Unknowns_(2013).pdf, 2013.

[27] ——, "End-point security failures, insight gained from secunia psi scans," in *Predict Workshop, February*, 2011.

[28] L. Bilge and T. Dumitras, "Before we knew it: an empirical study of zero-day attacks in the real world," in *Proceedings of the ACM conference on Computer and communications security (CCS)*, 2012, pp. 833–844.

[29] S. Beattie, S. Arnold, C. Cowan, P. Wagle, C. Wright, and A. Shostack, "Timing the application of security patches for optimal uptime." in *LISA*, vol. 2, 2002, pp. 233–242.

[30] M. D. Bond and K. S. McKinley, "Probabilistic Calling Context," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007, pp. 97–112.

[31] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang, "Precise calling context encoding," in *the 32nd International Conference on Software Engineering (ICSE)*, 2010, pp. 525–534.

[32] Q. Zeng, J. Rhee, H. Zhang, N. Arora, G. Jiang, and P. Liu, "DeltaPath: Precise and Scalable Calling Context Encoding," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2014, pp. 109–119.

[33] G. Novark, E. D. Berger, and B. G. Zorn, "Exterminator: automatically correcting memory errors with high probability," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2007, pp. 1–11.

[34] C. Kreibich and J. Crowcroft, "Honeycomb: creating intrusion detection signatures using honeypots," *Computer Communication Review*, vol. 34, no. 1, pp. 51–56, 2004.

[35] H.-A. Kim and B. Karp, "Autograph: Toward Automated, Distributed Worm Signature Detection," in *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, ser. SSYM'04. USENIX Association, 2004, p. 19.

[36] J. Newsome, B. Karp, and D. Song, "Polygraph: automatically generating signatures for polymorphic worms," in *IEEE Symposium on Security and Privacy (S&P)*, 2005, pp. 226–241.

[37] Z. Liang and R. Sekar, "Fast and Automated Generation of Attack Signatures: A Basis for Building Self-protecting Servers," in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, 2005, pp. 213–222.

[38] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez, "Hamsa: fast signature generation for zero-day polymorphic worms with provable attack resilience," in *IEEE Symposium on Security and Privacy (S&P)*, 2006, pp. 15–29.

[39] J. Newsome, D. Song, J. Newsome, and D. Song, "Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software," in *Proceedings of the 12th Network and Distributed Systems Security Symposium (NDSS)*, 2005.

[40] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt, "Automatic Diagnosis and Response to Memory Corruption Vulnerabilities," in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, 2005, pp. 223–234.

[41] J. Newsome, D. Brumley, and D. Song, "Vulnerability-Specific Execution Filtering for Exploit Prevention on Commodity Software," in *Proceedings of the 13th Symposium on Network and Distributed System Security (NDSS)*, 2006.

[42] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-end Containment of Internet Worms," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP)*, 2005, pp. 133–147.

[43] A. Smirnov and T. Chiueh, "Automatic Patch Generation for Buffer Overflow Attacks," in *Third International Symposium on Information Assurance and Security*, 2007, pp. 165–170.

[44] Z. Lin, X. Jiang, D. Xu, B. Mao, and L. Xie, "AutoPaG: Towards Automated Software Patch Generation with Source Code Root Cause Identification and Repair," in *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, 2007, pp. 329–340.

[45] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *31st International Conference on Software Engineering (ICSE)*, 2009, pp. 364–374.

[46] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *35th International Conference on Software Engineering (ICSE)*, 2013, pp. 802–811.

[47] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2016, pp. 298–312.

[48] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: treating bugs as allergies - a safe method to survive software failures," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005, pp. 235–248.

[49] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan *et al.*, "Automatically patching errors in deployed software," in *Proceedings of the ACM 22nd Symposium on Operating Systems Principles (SOSP)*, 2009, pp. 87–102.

[50] T. Ball and J. R. Larus, "Efficient path profiling," in *Proceedings of the 29th annual ACM/IEEE International Symposium on Microarchitecture (Micro)*, 1996, pp. 46–57.

[51] T. Mytkowicz, D. Coughlin, and A. Diwan, "Inferred call path profiling," in *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009, pp. 175–190.

[52] M. L. Seidl and B. G. Zorn, "Segregating heap objects by reference behavior and lifetime," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998, pp. 12–23.

[53] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," in *Symposium on Security and Privacy (S&P)*, 2003, pp. 62–75.

[54] N. Nethercote and J. Seward, "How to Shadow Every Byte of Memory Used by a Program," in *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE)*, 2007, pp. 65–74.

[55] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering (FSE)*, 2008, pp. 308–318.

[56] B. Buck and J. K. Hollingsworth, "An API for runtime code patching," *The International Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 317–329, 2000.

[57] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for evaluating bug detection tools," in *In Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[58] NIST, "SAMATE Reference Dataset," http://samate.nist.gov/SRD.

[59] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman, "The Matter of Heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, 2014, pp. 475–488.

[60] J. Wang, M. Zhao, Q. Zeng, D. Wu, and P. Liu, "Risk Assessment of Buffer Heartbleed Over-Read Vulnerabilities," in *the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2015, pp. 555–562.

[61] X. Chen, A. Slowinska, and H. Bos, "On the Detection of Custom Memory Allocators in C Binaries," *Empirical Software Engineering*, vol. 21, no. 3, pp. 753–777, 2016.