Towards Efficient Heap Overflow Discovery

Xiangkun Jia, *TCA/SKLCS, Institute of Software, Chinese Academy of Sciences;*
Chao Zhang, *Institute for Network Science and Cyberspace, Tsinghua University;*
Purui Su, Yi Yang, Huafeng Huang, and Dengguo Feng, *TCA/SKLCS, Institute of Software, Chinese Academy of Sciences*

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/jia

This paper is included in the Proceedings of the 26th USENIX Security Symposium

August 16–18, 2017 • Vancouver, BC, Canada

ISBN 978-1-931971-40-9

Open access to the Proceedings of the 26th USENIX Security Symposium is sponsored by USENIX

Towards Efficient Heap Overflow Discovery

Xiangkun Jia[1,3], *Chao Zhang*[2]✉, *Purui Su*[1,3]✉, *Yi Yang*[1], *Huafeng Huang*[1], *Dengguo Feng*[1]

[1]*TCA/SKLCS, Institute of Software, Chinese Academy of Sciences*
[3]*University of Chinese Academy of Sciences*
{jiaxiangkun, yangyi, huanghuafeng, feng}@tca.iscas.ac.cn purui@iscas.ac.cn

[2]*Institute for Network Science and Cyberspace*
Tsinghua University
chaoz@tsinghua.edu.cn

Abstract

Heap overflow is a prevalent memory corruption vulnerability, playing an important role in recent attacks. Finding such vulnerabilities in applications is thus critical for security. Many state-of-art solutions focus on runtime detection, requiring abundant inputs to explore program paths in order to reach a high code coverage and luckily trigger security violations. It is likely that the inputs being tested could exercise vulnerable program paths, but fail to trigger (and thus miss) vulnerabilities in these paths. Moreover, these solutions may also miss heap vulnerabilities due to incomplete vulnerability models.

In this paper, we propose a new solution HOTracer to discover potential heap vulnerabilities. We model heap overflows as *spatial inconsistencies* between heap allocation and heap access operations, and perform an in-depth offline analysis on *representative* program execution traces to identify heap overflows. Combining with several optimizations, it could efficiently find heap overflows that are hard to trigger in binary programs. We implemented a prototype of HOTracer, evaluated it on 17 real world applications, and found 47 previously unknown heap vulnerabilities, showing its effectiveness.

1 Introduction

Memory corruption vulnerabilities are the root cause of many severe threats, including control flow hijacking and information leakage attacks. Among them, stack corruption vulnerabilities used to be the most popular ones. As effective defenses [3, 12, 15, 19, 24, 35, 46, 47] against stack corruption are deployed gradually, nowadays heap overflow vulnerabilities become more popular. For example, it is reported that about 25% of exploits against Windows 7 utilized heap corruption vulnerabilities [28].

There are a lot of sensitive data stored in the heap, including heap management metadata associated with heap objects (e.g., size attributes, and linked list pointers), and sensitive pointers within heap objects (e.g., pointers for virtual function calls). It makes the heap a valuable target to attack. As the heap layout is not deterministic, heap overflow vulnerabilities are in general harder to exploit than stack corruption vulnerabilities. But attackers could utilize techniques like heap spray [16] and heap feng-shui [43] to arrange the heap layout and reliably launch attacks, making heap overflow a realistic threat.

Several solutions are proposed to protect heap overflow from being exploited, e.g., Diehard [4], Dieharder [34], Heaptherapy [52] and HeapSentry [33]. In addition to runtime overheads, they also cause denial of service, because they will terminate the process when an attack is detected. So it is imperative to discover and fix heap overflows in advance.

In general, both static analysis and dynamic analysis can be used to detect heap vulnerabilities. But static analysis solutions (e.g., [21, 36]) usually have high false positives, and are only fit for small programs. In addition to its intrinsic challenge (i.e., alias analysis), static analysis may generate false positives because the heap layout is not deterministic [27]. But for any specific execution, the spatial relationships between heap objects are deterministic. So, it's easier and more reliable to use dynamic analysis to detect heap vulnerabilities.

Online dynamic analysis is the mostly used state-of-art heap vulnerability detection solutions. In general, they monitor target programs' runtime execution (e.g., by tracking some metadata), and detect vulnerabilities by checking for security violations or program crashes. For example, AddressSanitizer [40] creates redzones around objects and tracks addressable bytes at run time, and detects heap overflows when unaddressable redzone bytes are accessed. Fuzzers (e.g., AFL [51]) test target programs with abundant inputs and report vulnerabilities when crashes are found during testing.

These solutions are widely adopted by industry to find vulnerabilities in their products. However, they all work in a *passive* way and could miss vulnerabilities. To report a vulnerability, they expect a testcase to exercise a vulnerable path and trigger a security violation. Even if a

passive solution could generate a bunch of inputs to reach a high code coverage and could catch all security violations, e.g., by combining AFL and AddressSanitizer, it could still miss vulnerabilities. For example, it may generate a bunch of inputs to exercise a vulnerable path, but fail to trigger the vulnerability in that path due to some critical vulnerability conditions.

Moreover, multiple vulnerabilities may exist in one program path. Passive solutions (e.g., fuzzers) may only focus on the first one and miss the others. For example, when analyzing a known vulnerability CVE-2014-1761 in Microsoft Word with our tool HOTracer, we found two new heap overflows, in the exact same program path which we believe many researchers have analyzed many times. It shows that, even for a vulnerable path in the spotlight, online solutions could not guarantee to find out all potential vulnerabilities in it.

On the other hand, *offline analysis* solutions could explore each program path thoroughly and discover potential heap vulnerabilities in a more proactive way, e.g., by reasoning about the relationship between program inputs and candidate vulnerable code locations. For example, DIODE [41] focuses on memory allocation sites vulnerable to integer overflow which will further lead to heap overflow, and then infers and reasons about constraints of the memory allocation size to discover vulnerabilities. Dowser [23] and BORG [31] focus on memory accesses sites that are vulnerable to heap overflow, and guide symbolic execution engine to explore suspicious buffer accessing instructions. However, neither of these solutions accurately model the root cause of heap overflow, and thus will miss many heap overflow vulnerabilities.

We point out that the root cause of heap overflow vulnerabilities is not the controllability of either memory allocation or memory access, but the *spatial inconsistency between heap allocation and heap access operations*. For example, if a program first allocates a buffer of size $x+2$, and then writes $x+1$ bytes into it, a heap overflow will happen if attackers make $x+2$ integer overflows but $x+1$ not. It is nearly impossible to identify this heap overflow vulnerability when only considering heap allocation or heap access operations.

In this paper, we propose a new *offline analysis* solution HOTracer, able to recover heap operations, check spatial consistency and discover heap overflow vulnerabilities. It first records programs' execution traces, no matter the corresponding inputs are benign or not. Then it recognizes heap allocation and heap access operation pairs and checks whether there are potential spatial inconsistencies. Furthermore, it checks whether the heap allocation and heap access operations could be controlled by attackers or not. If either one is controllable (i.e., tainted or affected by inputs), HOTracer reasons about the path conditions and spatial inconsistency to generate a PoC (i.e., proof-of-concept) input for the potential vulnerability.

In this way, our solution could discover potential vulnerabilities that may be missed by existing online and offline solutions. For online solutions, e.g., AFL and AddressSanitizer, they rely on delicate inputs to trigger the vulnerabilities. Our solution could work fine as long as the inputs could exercise any heap allocation and heap access operations.

On the other hand, a combination of existing offline solutions, e.g., DIODE and Dowser, seems to be able to achieve the same goal as HOTracer. However, the combination is incomplete. DIODE only considers heap allocation vulnerable to integer overflows, and Dowser only focuses on heap accesses via loops. Moreover, to make the combination practical and efficient in the real world, we have to solve several challenges.

First, there could be numerous execution traces to analyze. Since recording and analyzing a trace is time-consuming, we could not aim for a high code coverage. Instead, we analyze programs with representative use cases, and explore significantly different program paths.

Second, we need to identify all heap operations from the huge execution traces (without source code). Even worse, many programs utilize custom memory allocators and custom memory accesses. HOTracer utilizes a set of features to identify potential heap operations. Moreover, we need to group related heap allocation and heap access operations that operate on same heap objects into pairs. But the number of such pairs is extraordinary large. HOTracer reduces the number of pairs by promoting low-level heap access instructions into high-level heap access operations, and prioritizes pairs to explore pairs that are more likely to be vulnerable.

Finally, it is challenging to generate concrete inputs to trigger the potential vulnerability in a specific heap operation pair, especially in large real-world applications, due to the program trace size and constraint complexity. HOTracer mitigates this issue by collecting only partial traces and concretizes inputs that do not affect the vulnerability conditions.

We implemented a prototype of HOTracer based on QEMU and analyzed 17 real world applications. HOTracer found 47 previously unknown vulnerabilities, showing that it is effective and efficient in finding heap vulnerabilities. In addition to finding new vulnerabilities, HOTracer could also be used to help identifying the root cause of a vulnerability. As shown in Figure 1, HOTracer could be used to triage vulnerabilities in crashes (or security violations) generated by online dynamic analysis tools (e.g., fuzzers), or even further explore the same path to discover vulnerabilities that may be missed.

In summary, we have made the following contributions.

- We proposed a new offline dynamic analysis solution, which is able to discover heap vulnerabilities

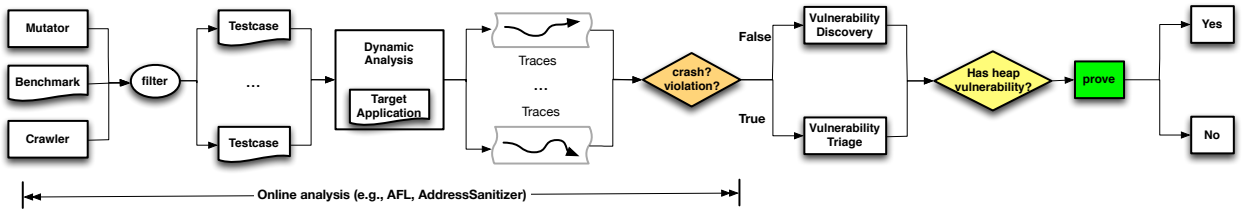


Figure 1: Applications of HOTracer. It relies on programs' execution traces, which can be generated in many ways, to discover heap vulnerabilities. It could discover heap vulnerabilities that are missed by online dynamic analysis tools (e.g., AFL and AddressSanitizer), because the testcases may not cause any runtime crashes or security violations at all, or only trigger shallow ones. It could also help clarifying the root cause (i.e., determine if it is a heap vulnerability or not) of a crash or violation.

that are hard to detect and prone to miss in benign traces, and able to help identifying the root cause of crashes and security violations in suspicious traces.

- We pointed out the root cause of heap vulnerabilities is inconsistency between heap operations. We also proposed a method to accurately model heap vulnerability conditions, with heap objects' spatial and taint attributes (i.e., affected by inputs or not).
- We addressed several challenges, including path explosion, pair explosion and constraint explosion, to make the solution practical and efficient.
- We implemented a prototype system, which is able to handle large real world applications and generate concrete inputs to prove heap vulnerabilities.
- We found 47 previously unknown vulnerabilities in 17 real world applications. Two of them are hidden in the same path as a known vulnerability.

2 Background

In this section, we will illustrate the root causes of heap overflow (and underflow) vulnerabilities, with a running example demonstrated in Figure 2.

2.1 Running Example

Usually, a heap access operation is performed via *a heap pointer* and a *memory access size*. In practice, the pointer used for heap access is usually derived from a heap object (e.g., `p1` at line 12 of Figure 2). As developers may use pointer arithmetic to get new pointers, we further decompose pointers into two parts: *pointer base addresses* and *offsets*. So a heap access operation is represented as $(ptr, offset_{ptr}, size_{access})$.

It is worth noting that, the offset and size may be derived from untrusted user input, either directly (e.g., the offset `size` at line 8) or indirectly (e.g., the size computed from the length of string `p1->name` at line 21).

On the other hand, a heap access operation's target object is represented by a memory range, i.e., an *allocation address* and *size*. We refer *obj* to *allocation address* of an object and represent it as $(obj, size_{obj})$.

```
1  #define SIZE (1024-4)
2  struct OBJ{
3    char name[SIZE];
4    void set_name(char* src, size_t size){
5      if(size > SIZE) exit(-2);
6      memcpy(name, src, size);
7      // off-by-one, when size == SIZE
8      name[size]=0;
9    }
10 };
11 int main(){
12   OBJ* p1 = new OBJ();
13   OBJ* p2 = new OBJ();
14   // tainted: size and input
15   input = get_input(&size);
16   // Vul #1: off-by-one if size=SIZE
17   p1->set_name(input, size);
18   // coalesce p1 and p2, causing p1 free.
19   free(p2);
20   // Vul #2: use after free
21   printf("p1 name: %s\n", p1->name);
22   return 0;
23 }
```

Figure 2: Two sample heap vulnerabilities: an off-by-one heap overflow and a use-after-free.

The allocation address is usually a heap address returned by memory management functions. However, the allocation size may be derived from user inputs. Even if it is not affected by inputs, e.g., developers use a constant number (e.g., 1020 at line 3) that seems to be big enough as the allocation size, the program may be still vulnerable to heap overflow.

Although experienced developers may sanitize inputs before using (e.g., line 5) to stop potential vulnerabilities, it is error-prone to implement such checks. For example, the check at line 5 misses one corner case where `size` equals to `SIZE`. This corner case will lead to an off-by-one vulnerability (i.e., a special heap overflow) at line 8, which is called at line 17. It will overflow one byte after the object `p1`, with value `0`.

Although this off-by-one vulnerability could only overwrite one extra byte of 0, it is still exploitable. For example, in the running example, it will cause a further use-

after-free vulnerability and lead to control flow hijacking. Details are omitted due to the space limitation.

2.2 Root Cause Analysis

The root cause of heap overflow (or underflow) is that, the heap access offset or size exceeds the target heap object's bound. More specifically, for a heap access via $(ptr, offset_{ptr}, size_{access})$ and target object $(obj, size_{obj})$, similar to related work SoftBound [30], we conclude that there is an underflow vulnerability if:

$$ptr + offset_{ptr} + size_{access} < obj.$$

Given that heap pointers *ptr* always refer to base objects' address *obj*, it equals to:

$$offset_{ptr} + size_{access} < 0. \quad (\mathbb{S}1)$$

There is an overflow vulnerability if:

$$ptr + offset_{ptr} + size_{access} > obj + size_{obj}.$$

i.e.,

$$offset_{ptr} + size_{access} > size_{obj}. \quad (\mathbb{S}2)$$

It is worth noting that, $offset_{ptr}$ may be a negative integer, but $size_{access}$ and $size_{obj}$ are always positive. If we use $offset_{ptr}$ as unsigned integer, and assume its bit-width is N, then Equation $\mathbb{S}1$ becomes

$$offset_{ptr} + size_{access} >= 2^{N-1}. \quad (\mathbb{S}3)$$

Moreover, $size_{obj}$ usually are smaller than 2^{N-1}. For example, objects on 32-bit platform usually are smaller than $2^{31} = 2G$ bytes. So, Equation $\mathbb{S}3$ implies Equation $\mathbb{S}2$. So, Equation $\mathbb{S}2$ always holds if there is a heap overflow or underflow.

In other words, a heap overflow or underflow exists if and only if:

$$range_{access} > range_{obj}. \quad (\mathbb{S})$$

where, $range_{access}$ represents $offset_{ptr} + size_{access}$, and $range_{obj}$ represents $size_{obj}$, and all values here are unsigned. Without loss of generality, we use the term heap overflow to represent both heap overflow and heap underflow in this paper.

Equation $\mathbb{S}$ depicts the inconsistency of spatial attributes between heap allocation and heap access. Our solution HOTracer uses it to build heap vulnerability conditions.

2.3 Observation

Even though no security violations are triggered by a benign input, potential vulnerabilities may still exist in the same program path. If user inputs could affect either heap allocation or heap access along this execution trace, they could change the spatial attributes of heap objects to satisfy Equation $\mathbb{S}$, cause spatial inconsistency between heap allocation and heap access, and thus trigger a heap overflow vulnerability.

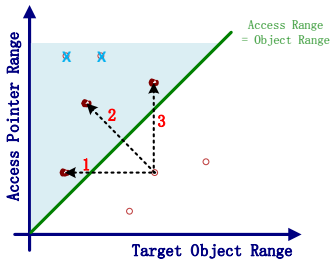


Figure 3: Heap overflow vulnerabilities exist in the shadow area, with the condition: $range_{access} > range_{obj}$.

We illustrate this possibility using Figure 3. If we can control the heap allocation size, we could make it smaller than the heap access size (e.g., dotted line 1 in the figure), to satisfy the Equation $\mathbb{S}$ and trigger heap overflows. If we can control both the heap allocation size and the heap access size, we could also make Equation $\mathbb{S}$ holds (e.g., dotted line 2).

3 Design

We aim to discover heap vulnerabilities with dynamic analysis, without relying on testcases to directly trigger vulnerabilities, and without source code. To achieve this goal, we analyze programs' execution traces offline, and explore potential vulnerable states along the binary traces.

Furthermore, to make the solution efficient and practical, we select representative testcases to generate a limited number of traces, perform *spot checks* on a small number of heap <*allocation, access*> operation pairs that are more likely to be vulnerable, and concretize values in path constraints and vulnerability constraints to speed up the constraint resolving.

3.1 System Overview

Based on the observation discussed in Section 2.3, our offline analysis tracks heap objects' spatial attributes (e.g., size) and taint attributes (e.g., affected by inputs or not, and affected by which input bytes) along the target execution trace.

Figure 4 shows an overview workflow of our solution HOTracer. It first pre-processes the sample inputs by first selecting representative inputs, and then feeds them into a dynamic analysis component to generate execution traces for each input. For a given trace, HOTracer traverses it offline again to do some in-depth analysis.

Then, it identifies heap allocation and heap access operations, and builds the heap layout. It also groups heap operations that operate on same objects into pairs.

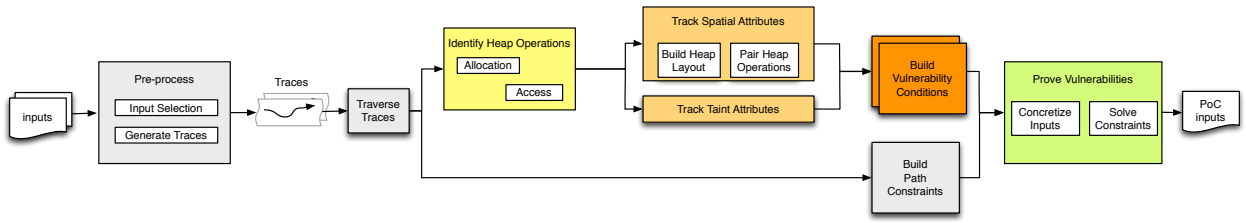


Figure 4: Overview of HOTracer's solution. It selects useful testcases and generate traces for each of them. Then it recognizes heap operations in traces, tracks heap objects' attributes and infer vulnerability conditions for each pair of heap operations. It finally generates proof-of-concept (PoC) inputs to prove vulnerabilities by reasoning about vulnerability conditions and path constraints.

Next, HOTracer tracks heap objects' spatial and taint attributes during execution traces. Based on these attributes, it builds the vulnerability conditions using Equation $\mathbb{S}$ for each pair of heap <*allocation, access*> operations.

Finally, it solves the vulnerability conditions, along with the path constraints, to check potential heap overflows, and generates concrete inputs to prove the existence of them.

Following this process, we figure out there are many challenges when making it work for real world applications, especially the usability and efficiency of this solution. First, there would be numerous execution traces to analyze. Second, there would be a large number of heap <*allocation, access*> operation pairs in each execution trace. Third, the path constraints and vulnerability condition constraints would be very large and complex to solve, especially for real world applications. In the remaining of this section, we will discuss our design choice to address these challenges.

3.2 Trace Generation Optimization

3.2.1 Testcase Selection

We may have too many input samples to analyze, and analyzing a single program trace thoroughly is expensive. On the other hand, many samples may exercise the same program path, and thus it is not necessary to analyze all of them. To mitigate this issue, we will only select representative inputs to analyze.

We use different heuristics to select seed inputs based on types of inputs. For known file types (e.g., multimedia input files), we crawl some sample inputs from the Internet. Then we parse the structure of these sample inputs, and utilize the file format information to select representatives from each sub-type (e.g., `tags` in `MP4` files). In general, we will perform a min-set coverage analysis to select a minimal set of testcases that covers all the subtypes. Based on the trivial knowledge that different subtypes of inputs will exercise different program paths, we could get a set of representative execution traces.

For unknown file types, we use fuzzers to generate a set of seed inputs, and distill the inputs to a minimum set which covers most code blocks. In this way, we could also get a set of representative testcases.

3.2.2 Trace Record and Replay

For each selected input, we need to feed it to the target program, and get its runtime execution trace for further offline analysis. It is critical to record the trace in a timely manner. Otherwise, it may cause timeout issues and interrupt the program execution.

We adopted the record-replay mechanism introduced in PANDA [18] to generate traces with a low overhead. In general, it has two phases to generate traces. In the *record* phase, it takes a snapshot of the system before execution, and records only changes at runtime. In this way, the recording process costs low overheads. In the *replay* phase, it interprets the snapshot and records to recover the full execution trace for further offline analysis.

3.3 Heap Operation Model

3.3.1 Heap Allocation Recognition

Heap objects are created by allocation functions. By analyzing heap allocation functions, we can get the size and address of heap objects, and update the spatial attributes of heap objects.

However, it is challenging to recognize all heap allocation functions accurately. In addition to standard APIs (e.g., `malloc` and `free`), developers usually develop custom heap allocators for different purposes. For example, Firefox uses a custom heap management implementation `Jemalloc`, to solve its memory fragmentation problems. We studied some popular custom allocators (e.g., `Jemalloc`, `Tcmalloc`, `MMgc`), and figured out their work flows share the same pattern as shown in Figure 5, and they have the following features.

First, the most important feature is the return values of memory allocators must be heap pointers.

A. An allocator always returns a pointer to the heap region, which is known for a specific platform.

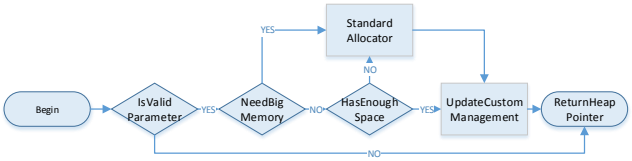


Figure 5: High-level work flow of custom allocators.

Second, the allocation size processing is also an important feature. It affects the memory allocation in several ways.

B1 Custom allocators have to use standard allocation interfaces to get memory from system when the allocator is called for the first time, or when the internal reserved memory pool is drained.

B2 Allocators usually keep different memory pools for different allocation sizes to improve allocation efficiency and ease the burden of boundary check.

B3 Allocators usually pad extra bytes at the end of objects to make objects aligned (with 4 bytes, 8 bytes etc.).

B4 Allocators usually maintain internal heap management structures and update them when allocating. To avoid concurrency issues, the heap allocators will lock the internal metadata before updating, e.g., by calling `EnterCriticalSection` on Windows platforms.

Third, memory allocation functions will be used by the program in special ways.

C1 The return value of an allocator will be first used in memory write operations before any read operations.

C2 A memory allocator will usually be invoked several times in a specific execution trace.

C3 The allocator will return different values in different invocations in most cases, unless the underlying memory is released before allocation.

C4 Some heap allocation functions will initialize the objects (e.g., set to 0) before returning, to avoid potential bugs (e.g., use of uninitialized variables).

We first identify all functions satisfying feature `A`. Then we point out ones satisfying at least one feature of `B1, B2, B3, B4`. Finally, we recognize ones satisfying at least one feature of `C1, C2, C3, C4`. In this way, we could get a set of candidate heap allocators. Furthermore, we will remove wrapper functions from the set.

It is worth noting that, identifying heap allocators in this way may generate false positives and thus increase the number of candidate pairs. From our evaluation, the false positive ratio is very low. On the other hand, this solution in general will not generate false negatives. So it will not prevent us from discovering potential heap vulnerabilities.

It is an open challenge to accurately identify all heap allocators in binary programs. Existing works like MemBrush [13] provide promising alternatives. MemBrush uses features A and C1, together with some other minor features to identify candidate allocators. The major difference is that, MemBrush uses dynamic online analysis to repeatedly invoke and test each candidate allocator with different parameters. However, the dynamic testing process is slow, and its accuracy improvement over our solution is not significant. So we only use the features proposed here to do a quick recognition.

3.3.2 Heap Operation Pairs

After recognizing heap allocators, we could recover the address and size attributes of heap objects and pointers, and update them along the execute trace. We further recover the heap layout with these spatial attributes, and maintain the *point-to* relationship between heap objects and pointers. So we could group heap allocation and heap access operations into pairs.

We also track the taint attributes of heap objects and pointers using taint analysis. Further we could check heap operations pairs that could be controlled by attackers for potential vulnerabilities.

3.4 Candidate Pair Reduction

There are too many heap allocation sites and heap access operations even in one single trace, making the number of candidate vulnerable pairs too large to analyze. As a result, it is crucial to reduce the number of candidate pairs.

We first abstract low level heap access instructions to high level operations to reduce the number of heap access operations, and then prioritize candidate pairs based on the likelihood of vulnerability in each pair. In this way, we could limit the number of candidate pairs to a reasonable number, and make further vulnerability discovery practical.

3.4.1 Heap Access Abstraction

We could easily recognize heap access *instructions* in the trace after recognizing all heap pointers and heap objects. A straightforward solution would be treating each heap access *instruction* in the traces as a heap access operation, and generating the pairs. However, it will explode the number of heap operation pairs. For example, a buffer copy could be compiled into a simple loop, or a `REP`-prefixed instruction, which is represented with a sequence of memory access instructions in the trace. Each iteration of every heap access instruction will contribute to a new heap operation pair. So the number of pairs grows rapidly in this way.

Obviously, we should treat each one of such loops and sequences of memory access as one heap access if possible, in order to reduce the number of heap operation pairs without missing any potential vulnerabilities.

On the other hand, it is also helpful to recover high-level heap access operations for other purposes. For example, it could help us to identify the size of heap access and the taint attributes of heap access operations.

Thus, we abstract heap access operations in the following order to reduce the number of heap operation pairs.

D1 We first recover simple loops that are used for heap access. We treat each occurrence of these loops in the trace as one heap access operation, but not any instruction within these loops.

D2 We then treat each sequence of heap access instructions that corresponds to one `REP` instruction in the trace as a single heap access operation.

D3 We finally treat every remaining instruction in the trace that accessed the heap as a heap access operation.

3.4.2 Heap Operation Pairs Sorting

The heap access abstraction phase greatly reduces the number of candidate heap <*allocation, access*> operation pairs. However, the number would be still big. We further mitigate this issue by prioritizing the heap operation pairs. Pairs that are more likely to be vulnerable will be explored first in the following steps.

First, we prioritize pairs that have access operations of type D2, since it is the most common case of heap buffer access operations. Then we prioritize pairs that have access operations of type D1.

Second, we will prioritize heap operation pairs depending on the ability of attackers, i.e., how well they could affect the heap operations. As shown in Figure 3, attackers may have different abilities to control heap operations. The order we use to prioritize these pairs is as follows.

E1 The heap allocation and heap access size are affected by *different* input bytes. It means attackers could change the element of heap pairs independently and make it inconsistent. This type is most vulnerable according to our experience.

E2 Only the heap allocation but not the heap access operation is affected by input bytes. This is also a popular case of heap overflow vulnerabilities. The famous IO2BO vulnerability [41, 53] in general falls into this category.

E3 Only the heap access but not the heap allocation operation is affected by input bytes. It is also vulnerable in this case if the access size exceeds the (constant) allocation size.

E4 The heap allocation and heap access size are affected by *same* input bytes. This type of heap operation happens a lot in practice. For example, the program allocates `X` bytes and later tries to access only `X` bytes. In most cases, this type is not vulnerable.

E5 Neither the heap access nor the heap allocation operation is affected by input bytes. Usually there should be no heap overflow in this case, unless there is a careless bug, e.g., the program allocates 100 bytes and tries to access 101 bytes no matter what inputs are given. Most tools are able to detect this kind of vulnerability.

Furthermore, considering the ability of constraint solvers, we will prioritize pairs that have simpler program path constraints, simpler computation of heap operation sizes, and shorter distance from allocation to access operations. This prioritization enables us to explore simpler pairs first and reason about them to discover vulnerabilities.

3.5 Constraint Solving Optimization

After getting the candidate vulnerable heap operation pairs, we could reason about each pair to confirm whether it is vulnerable or not. Basically, we will collect the path constraint and the vulnerability condition for each candidate pair, and then query the constraint solver to generate PoC if possible.

However, the program path and vulnerability condition constraints may be too complex for solvers to resolve. We thus proposed several optimizations to mitigate this issue.

First, HOTracer will concretize irrelevant bytes in the constraints. More specially, only bytes occurring in the vulnerability conditions will be marked as symbolic, other bytes will be replaced with concrete values used in current execution trace. So we only need to solve parts of the constraints.

Moreover, HOTracer only collects instructions from the first related input point till the vulnerability points, and performs symbolic execution on them. In this way, it could greatly reduce the possibility of solver failure or timeout.

4 Implementation

In this section, we will discuss implementation details of HOTracer, and our practical experience with real world programs. Our current prototype focuses on analyzing Windows x86/x64 applications. But the techniques we developed are general, and could be extended to other platforms.

4.1 Collect Traces

HOTracer relies on program execution traces to discover heap vulnerabilities. The diversity of traces affect the

number of program paths that will be analyzed. To generate traces, we first select input testcases for target programs, and then test them on target programs and record their runtime executions.

4.1.1 Testcase Selection

The testcases could origin from different sources, e.g., fuzzers, existing benchmarks, or network crawlers, as shown in Figure 1.

As discussed in Section 3.2.1, we will use parsers to parse testcases of known input format, and select representative testcases based on their sub-types. In general, we trivially perform a min-set coverage analysis to select a minimal set of testcases that cover all the sub-types. We also use fuzzers to generate testcases of unknown input format, and further distill the testcases based on their code coverage information.

4.1.2 Trace Generation

Given an input testcase, we use our previous dynamic analysis framework [32] to generate the execution trace. Our dynamic analysis framework implements a record-replay mechanism similar to PANDA [18], based on the open-source whole-system hardware emulator QEMU, to improve the performance of recording.

It takes a snapshot of the system before execution, and records changes to the CPU state and memory in a changelog file during the execution. In this way, it will not slowdown QEMU much. After the runtime execution finished, we could replay the snapshot and changelog file to generate an execution trace, which is identical to the runtime execution trace.

4.2 Identify Heap Operations

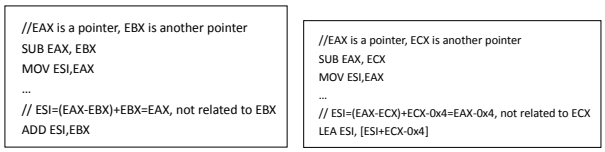
Given the trace, we need first identify heap allocation and heap access operations, as well as the heap objects and pointers in the trace, to detect potential heap overflow vulnerabilities.

4.2.1 Heap Allocation Recognition

Based on the heuristics described in Section 3.3.1, we could identify most custom heap allocators with a high accuracy. After identifying these heap allocators, we could identify the sizes and addresses of heap objects along a trace, from the arguments and return values of these allocators. Furthermore, by performing data flow analysis, we could recognize all heap pointers and their mappings to heap objects (described in Section 4.3).

4.2.2 High Level Heap Access

Rather than checking every heap access instruction in the trace, we check some high level heap access operations first, to reduce the number of candidate heap operation pairs.



(a) add (b) lea

Figure 6: Corner cases of taint propagation.

Heap Access of Type D1 It is common for developers to use a loop to access a heap object. This is also a common source of heap overflow vulnerabilities. A loop in the trace is a continuously repeated sequence of instructions ending with jump instructions, unlike its representation in the control flow graph (i.e., a backward edge) [8]. HOTracer takes the sequence of instruction addresses in the trace as a string, and identifies loops by searching for continuously repeated sub-strings.

We record instructions in a historybuffer. While analyzing an instruction in the binary trace, we look forward in the historybuffer for the appearance of this instruction. If it is in a loop, the following instructions would repeat the sequence between this instruction and the last one. The sequence is the loop body and jump instructions at the end of the loop body infer the exit conditions. In this way, we could identify loops with one sequential scanning.

For each loop, we also care about whether its execution is affected by inputs. We infer the relationship between the count of loop iterations and inputs at the loop exit point (i.e., a conditional branch instruction). We also count the number of iterations in current trace to infer the access range of a loop.

Our current prototype could find nested loops but not complicated overlapped loops [45, 50]. We leave it as a future work.

Heap Access of Type D2 It is easy to recognize `REP` instructions in a trace, so does the access size (i.e., `ECX`), by comparing the instruction sequence in the trace with instructions in the original binaries.

4.3 Track Spatial Attribute

4.3.1 Build Heap Layout

From the execution trace, we know the exact values of heap pointers and addresses of objects, and thus we could easily get the layout of the heap. During the data flow analysis, we track heap objects' *spatial* attributes according to allocation operations. The attributes are initialized to allocation sizes when objects are allocated. When deallocating, their spatial attributes are updated to 0. In this way, we could get the heap state at any moment.

4.3.2 Pair Heap Operations

As we pointed in Section 2.2, we analyze heap operations in pairs based on the point-to relationships between pointers and objects. However, it is not trivial to infer whether a pointer should point to an object, because the pointer value may exceed its expected object's memory region (e.g., due to heap overflow). In other words, we could not rely only on the values of heap pointers and addresses of objects.

So we perform an analysis to track pointers' *provenances*, in order to build the accurate heap layout. In general, when a pointer is set to point to an allocated object, we set the base object as this pointer's provenance. This provenance will be propagated along the program trace, e.g., via pointer arithmetic operations, to other pointers. The provenance of a pointer will be updated when it is reset. This provenance analysis is similar to classic taint analysis [6].

As a result, by querying a pointer's provenance, we could always infer which object it should point to, even if the pointer's value is overflowed. For the target object at access points, we could easily get its allocation operation. In this way, we group specific heap access and heap allocation operation into a pair, and further evaluate their spatial attributes to discover potential heap overflows.

Under-taint Issue: However, there is a corner case during the taint (i.e., provenance) propagation for pointers. For example, the `SUB` instruction in Figure 6a will usually clean the taint attribute of the pointer (i.e., `EAX`), since the result is constant and is not a pointer any more [6]. However, this offset could be later used to compute another pointer (e.g., the final `ESI` register) which should be tainted. As a result, the new pointer will take a wrong attribute from `EBX` rather than `EAX`.

We propose a new solution to mitigation this issue. More specifically, we tag the destination register `EAX` of the `SUB` instruction with a set of taint attributes $(provenance_{EAX}, -provenance_{EBX})$. It is worth noting that, objects' addresses (i.e., pointers' provenances) are usually lower than a specific value on a given platform, so $-provenance_{EBX}$ is different from any normal taint attribute. We can detect this abnormal attribute when it is used in instructions like `ADD` and `LEA`, and recover the correct taint attribute of operands.

4.4 Track Taint Attribute

HOTracer tracks *taint* attributes of different values (e.g., sizes of heap objects, offsets of heap pointers etc.). In general, it performs a fine-grained taint propagation analysis to track each value's source (i.e., specific bytes in the input).

We use the same taint analysis solution as previous provenance analysis, to track values' taint attributes, i.e., which input bytes affect the target values. What's different is that, we use the position of input bytes as taint attributes, and propagate these attributes along the trace.

However, sometimes the inputs will not directly affect values used in heap access or allocation operations. For example, programs may use `strlen` or other custom functions to infer some values (e.g., length) of the inputs, and use them as size to allocate memory. In this case, the heap allocation is indirectly affected by the inputs. These inferred values are *control dependent* on the inputs. For example, the return value of `strlen` control-depends on the input string, i.e., whether the input character equals to '\0' or not.

Classical dynamic taint analysis solutions usually will not propagate taint information for control dependencies [39], due to the concern of taint propagation efficiency. Instead, HOTracer performs an extra backward analysis, to search for the definition points of heap allocation sizes. Given the high-level loop and branch information we recovered, if we find out the definitions are control-dependent on the inputs, we will mark the allocation sizes as tainted.

Furthermore, the traces we collected only include user-space instructions. So some data flow will be missing when the kernel kicks in. HOTracer will check value of registers before and after the executed instructions (e.g., `sysenter`). If the value of any register other than the destination operand has changed, a potential data flow missing is found. In this case, we will clean the taint attributes of the registers, to avoid false positives. Another choice is using summary information of syscalls, to propagate the taint attributes for kernel execution. However, it requires a lot of engineering work to correctly summarize the side-effects of all syscalls.

4.5 Build Vulnerability Condition

For each pair of heap access and heap allocation operations, we assume the heap access has attributes $range_{access}$ and the target heap object has attributes $range_{obj}$,

1. a heap overflow exists if $range_{access} > range_{obj}$ is true for current testcase, i.e., Equation $\mathbb{S}$ holds.
2. a potential heap overflow exists if either $range_{access}$ or $range_{obj}$ is tainted, which may make Equation $\mathbb{S}$ hold.

In case 1, we can confirm the existence of heap overflow in current trace and show the root cause of heap overflows. In case 2, we could infer the conditions of potential heap vulnerabilities and reason about these conditions. More specially, by performing symbolic execution, we can build the constraints between input bytes and the spatial attributes of $range_{access}$ and $range_{obj}$. Together with the vulnerability Equations $\mathbb{S}$, we can build the vulnerability conditions.

First, for a heap access operation, if the heap access' range is larger than the object's range, then a heap overflow is confirmed. It means the original input already triggers the vulnerability.

It is worth noting that, heap management functions will also access heap objects' metadata that are out of the objects' bound. But these overflow access operations are benign. So we will rule out these heap access when discovering heap vulnerabilities.

Second, for a heap access, if the heap pointer's size is not larger than the object's size, we will discover potential heap overflow by checking their taint attributes. There are also three cases: (1) if only the object's size or the pointer's size is tainted (e.g., line 1 and line 3 in Figure 3), there may be a heap overflow; (2) if both the object's size and pointer's size are tainted (e.g., line 2 in Figure 3), there may be a heap overflow too. (3) neither the object's size nor the pointer's size is tainted, then there are no heap overflow in this heap access, unless there is an instinctive bug that could be triggered no matter what inputs are given.

For the case (2), i.e., when the object's size and pointer's size are both tainted, there are also two subcases: they rely on different input bytes; or they rely on same input bytes. In the former case, usually there will be some heap overflows. In the latter case, there may be no heap overflows at all. For example, if we allocate a heap buffer with size `X` from input, and access heap with size `X`, then there are no overflows. But vulnerabilities like integer overflow may cause the allocation size mismatches with the access size, even though they rely on same inputs. HOTracer will check the existence of integer overflow in this case (i.e., IO2BO [41, 53]).

4.6 Prove Heap Vulnerabilities

After building the vulnerability conditions, the last step is to find concrete inputs that trigger the vulnerabilities. We use the widely used constraint solver Z3 [17] to resolve the constraints and generate inputs.

4.6.1 Build Path Constraints

Inputs satisfying only the vulnerability conditions may not trigger vulnerabilities at all, since (1) it may not reach the vulnerable point that we analyzed, because a different program path is exercised, and (2) it may be blocked by some input validations deployed in the program.

So, HOTracer will also collect the program path constraints, i.e., how the input bytes affect the branches in the trace. By feeding the vulnerability conditions and program path constraints to the solver, we could get inputs that will exercise the same path and trigger heap vulnerabilities, or confirm that there are no heap vulnerabilities along this path, or fail because the state-of-art solvers could not solve the constraints.

4.6.2 Constraint Simplification

As discussed in Section 3.5, HOTracer will only collect path constraints related to bytes used in the vulnerability conditions, and use concrete values for other input bytes used in the path constraints, to simplify the path constraints.

We also notice that, programs may read the same input bytes multiple times via multiple functions. For example, some programs use the first read operation for preproccessing, and a second read to process the content. To reduce the complexity, we will only collect path constraints from the last relevant read to the vulnerability points. If inputs generated from these constraints could not trigger the vulnerability, then we will include path constraints starting from previous reads.

4.6.3 Mutate and Verify

Since our vulnerability conditions only consider heap overflow, the concrete inputs generated by constraint solvers (called *candidate PoC inputs*) may not trigger crashes or other severe consequences.

On the other hand, inputs that could trigger crash would make further analysis easier, e.g., debugging and bug fixing. So, HOTracer performs another step to filter the candidate PoC inputs, to find out inputs that could trigger crashes.

The idea is that, we will compare the candidate PoC input with the seed input, to find out the input bytes that have changed. Then we use a simple fuzzer to mutate only these bytes with simple mutation strategies, e.g., minimum and maximum signed or unsigned integers, common values like 0 and 1, and random bytes etc. We will test each mutated input, to see whether it could trigger a crash.

For any candidate PoC input, if one of its mutations triggers a crash, HOTracer will report a heap vulnerability together with this mutation input. Otherwise, HOTracer will ignore this candidate PoC input.

It is worth noting that, the ignored candidate PoC inputs may still be valuable. The associated vulnerability instructions could be exploited with advanced exploits. We leave it as a future work to analyze these candidate PoC inputs and whether they are exploitable.

5 Evaluation

In this section, we present the evaluation of our solution HOTracer. Our prototype implementation is based on our existing QEMU-based dynamic analysis framework. The seed selection component takes about 40 LOC of shell scripts, the heap operation identification component takes about 1.3K LOC of C++ code, the heap attributes tracking and vulnerability condition building components take about 8K LOC of C++ code, and the vulnerability proving component takes about 9K LOC of C++ code.

Table 1: Zero-day vulnerabilities found by HOTracer.

ID (count)	Application	version	input	bug status
new (1)	Feiq	3.0.0.2	tcp	reported
new (1)	WMPlayer	12.0.7601	mp4	reported
new (3)	VLC	2.2.1	mp4	fixed
new (1)	VLC	2.2.4	mp4	reported
new (2)	iTunes	12.4.3.1	mp4	reviewing
new (1)	ffmpeg	c0cb53c	mp4	CVE
new (6)	QQPlayer	3.9(936)	mp4	rewarded
new (1)	QQMusic	11.5	m4a	rewarded
new (1)	BaiduPlayer	5.2.1.3	mp4	reviewing
new (2)	RealPlayer	16.0.6.2	mp4	CVE
new (1)	MPlayer	r37802	mp4	reported
new (3)	KMPlayer	3.9.1.138	mp4	fixed
new (4)	KMPlayer	4.1.1.5	mp4	reported
new (7)	Potplayer	1.6.60136	mp4	fixed
new (2)	Potplayer	1.6.62949	mp4	reported
new (5)	Splayer	3.7	mp4	reported
new (2)	MS Word	2007,10,16	rtf	reviewing
new (1)	WPS Word	10.1.0.5803	doc	reported
new (2)	OpenOffice	4.1.2	doc	reviewing
new (1)	IrfanView	4.41	m3u	fixed

The analysis environment is a Ubuntu 12.04 system running on a computer with 12G RAM and Intel Xeon (R) CPU E5630 @ 2.53GHz*8.

5.1 Effectiveness

Table 1 shows previously unknown vulnerabilities found by HOTracer in our experiment. The target applications we tested are popular applications in Windows 7 operating system, including word document processing applications Microsoft Word and OpenOffice, video players KMPlayer and potplayer, and photo viewers IrfanView etc.

These applications are tested within QEMU, with some selected testcases (Section 5.5). The traces collected by QEMU are then analyzed by HOTracer. Although we only demonstrated Windows applications here, the solution we proposed could be extended to other platforms (including Linux on x86, Android on ARM), since they are both supported by QEMU and our solution is general.

As shown in the table, we have found 47 previously unknown vulnerabilities in 17 applications (of latest versions). All vulnerabilities are validated with proof-of-concepts (i.e., PoC) inputs that could trigger crashes.

It is worth noting that, all vulnerabilities here are investigated manually and confirmed to be unique. We use two factors to distinguish vulnerabilities, i.e., the overflow instructions along with the call context, and the key input bytes' roles (i.e., structure fields) in the input structures.

5.2 False Negatives and False Positives

In general, it is impossible to evaluate false negatives of a vulnerability detection solution, since we do not have the ground truth of how many vulnerabilities exist in programs.

Table 2: Known heap overflow vulnerabilities replayed and validated by HOTracer.

ID	Application	version	input
CVE-2010-1932	Xnview	1.97.4	mbm
CVE-2011-5233	irfanview	4.30	tif
OSVDB-83812	ZipItFast	3.0 pro	zip
CVE-2014-1761	Microsoft Word	2010	rtf
EDB-ID-39353	VLC	2.2.1	mp4
EDB-ID-17363	1ClickUnzip	3.0.0	zip
CVE-2010-2553	MediaPlayer	9.00.00.4503	avi
CVE-2015-0327	Adobe Flash	13sa	swf

Instead, we chose several known vulnerabilities and their corresponding program paths as a ground truth. To evaluate the false negatives, we gave some *benign* testcases that exercise the same program paths as the target vulnerabilities to our tool, and then used it to analyze these programs. It is worth noting that, in this experiment, HOTracer does not have any other knowledge of the vulnerabilities, except the benign testcase.

Table 2 shows 8 known vulnerabilities in 8 applications. HOTracer is able to discover 6 of them on its own, except vulnerabilities CVE-2015-0327 and CVE-2010-2553.

For vulnerability CVE-2015-0327 in Adobe Flash, it requires to override a standard API, causing it behave differently at heap allocation site and heap access site. Currently, our solution could not build and solve this type of constraints. For vulnerability CVE-2010-2553 in Media Player, our prototype system missed a type-D1 heap access (i.e., a loop), but only paid attention to one type-D2 heap access (i.e., a REP instruction) inside this loop. It shows that it is necessary to further improve our loop recognition algorithm to deal with complex real world applications. However, HOTracer could validate these two vulnerabilities if given the correct PoC samples.

More interestingly, when analyzing the program path (with a benign input) of the vulnerability CVE-2014-1761 in Microsoft Word 2010, HOTracer found two new vulnerabilities, which even affect the latest version of Microsoft Word. We believe for known vulnerabilities like CVE-2014-1761, vendors and researchers have already performed many thorough testings. It thus shows that even for known vulnerabilities in spotlight, existing solutions may still miss potential vulnerabilities.

On the other hand, since HOTracer only reports vulnerabilities with proof-of-concept (PoC) testcases that could trigger crashes, there are no false positives. However, it is possible that some reported vulnerabilities are not exploitable.

Table 3: Metrics of the analysis performed by HOTracer, including the size of snapshot, changelog, traces and constraints, the instruction count in the traces, and the time spent to record, replay, analyze and extract traces.

ID	record-replay phase				analysis phase			resolve phase		
	snapshot size	changelog size	record time	replay time	trace size	trace #instr.	analy. time	relev. #instr.	constaint file size	extr. time
CVE-2010-1932	430.6MB	36.6MB	29s	486s	2.8GB	12.3M	99s	619	192KB	37s
CVE-2011-5233	516.1MB	18.5MB	37s	738s	9.8GB	43.9M	112s	795	251KB	96s
OSVDB-83812	819.3MB	13.6MB	83s	1257s	31.9GB	142.5M	787s	10	4.3kB	52s
CVE-2014-1761	855.3MB	52.3MB	178s	3712s	205.8GB	918.6M	6478s	183	17.8KB	198s
EDB-ID-39353	507.6MB	15.0MB	62s	271s	8.2GB	36.7M	10s	3082	331.1KB	674s
EDB-ID-17363	500.2MB	32.6MB	70s	889s	3.1GB	13.7M	45s	2313	191.2KB	502s
CVE-2010-2553	282.5MB	22.9MB	100s	806s	10.7GB	47.6M	565s	-	-	-
CVE-2015-0327	610.8MB	13.8MB	34s	682s	25.2GB	112.4M	709s	-	-	-

5.3 Bug Reports

We reported all the new vulnerabilities to vendors, and most vendors are very active in responding. As shown in Table 1, three vendors have completely fixed their products, i.e., IrfanView, ffmpeg and Realplayer. Two of them have already been assigned with CVE ID [1].

During our experiments, we found that some tested programs (e.g., VLC, KMPlayer) have released new updates for the bugs we reported. We then applied HOTracer to the latest version and found that some vulnerabilities still exist. In other words, three vendors have only partially fixed their products.

Moreover, vendors of QQPlayer and QQMusic have confirmed the vulnerabilities in their products and rewarded us for the report. Five other vendors including Microsoft, Apple, and OpenOffice are still reviewing the issues.

In summary, vendors are willing to fix security bugs in their products. However, during the communication, we also found that vendors were more willing to fix vulnerabilities which are sure to be of high risk or exploitable. For vulnerability reports with only PoC crash inputs, they do not take it seriously, especially for larger companies. We believe one reason of this kind of negative attitude is that, there are too many vulnerability reports waiting in their pipelines. In other words, there are too many vulnerabilities (or bugs) in daily applications, calling for solutions like HOTracer to help.

Due to the time limit, we only manually checked 10 of these vulnerabilities to see whether they are exploitable. In general, it is challenging to conduct exploits against a vulnerability, especially in the context of defenses deployed in modern platform. We found 9 of them are likely exploitable.

5.4 Efficiency

As we discussed, HOTracer first takes a snapshot of the system, and then records changelog during execution. After that, HOTracer replays the snapshot and changelog to generate traces, and then performs analysis on the traces to find potential heap vulnerabilities. Finally, it extracts relevant instructions from the trace and build the constraints to generate concrete inputs to prove vulnerabilities. Table 3 shows some detailed metrics of these different phases.

As we can see from the table, the record-replay mechanism works well. It will not break the target applications' functionality, e.g., causing program timeout due to a heavy runtime monitoring or recording. The traces of real world applications are usually very large (e.g., 205GB for CVE-2014-1761 in Microsoft Word), much larger than the snapshot and changelog size. It also takes much longer (e.g., 20 times longer) time to replay and generate the traces than recording only changelogs.

HOTracer performs the offline analysis, including heap attributes tracking and vulnerability modeling, on the traces to discover heap vulnerabilities. As shown in the table, the offline analysis time is close to the replay time, varying from 2 minutes to 50 minutes. The analysis time depends on the number of instructions in the traces. The more instructions a trace has, the more time the analysis needs.

After identifying potential heap vulnerabilities, HOTracer will extract instructions relevant to the candidate vulnerabilities and build the constraint files to query constraint solvers (e.g., Z3). As shown in the table, it requires 0.5 to 15 minutes to extract the related instructions and build constraints, depending on the number of relevant instructions.

And for the vulnerability CVE-2015-0327 in Adobe Flash and CVE-2010-2553 in Media Player, our prototype fail to find out the vulnerability. So we do not have any data for its resolving phase in the table.

5.5 Testcases Selection

The testcases we use will affect the vulnerability assessment HOTracer could provide to target applications. In addition to utilizing fuzzing tools like AFL to generate

[1] CVE-2016-6164 for ffmpeg, CVE-2016-9931 for RealPlayer

testcases, we also crawl existing databases to find testcases.

In our experiment, we searched a *public* database [2] of multimedia files with more than 10,000 testcases. Among them, there are more than 800 MP4/MOV files in the database. All of them contain the tag (i.e., sub-type) `moov`, and only a few files have the tag `avcC` and `trun`. By parsing the structures of these files and performing a min-set coverage analysis, we reduce the number of files to 20, without losing the path coverage.

5.6 Details of Trace Analysis

Table 4 and 5 show some detail evaluations of our trace analysis. Due to the space limitation, only parts of the results are shown in these tables.

After pre-processing, HOTracer selects input testcases and generates corresponding program traces. As shown in the table, the traces' sizes are relatively large, but HOTracer could still analyze them.

HOTracer groups heap allocation and heap access operations into pairs to discover potential heap vulnerabilities. Each pair operates on a same heap object, and is related to a set of input bytes. The number of heap operation pairs is thus critical to the efficiency of our solution. We will further demonstrate that the optimizations we performed greatly reduced the number of heap operation pairs.

Accuracy of Heap Allocation Recognition. The accuracy of the heap allocation affects the number of heap operation pairs. Table 4 shows the accuracy of our recognition algorithms.

On the left of the table, it shows some statistics of some sample traces that are analyzed, including the snapshot and record size, as well as the record and replay time.

On the right of the table, it shows the time cost to identify these heap allocators. In general, the identification time is related to the trace size. A larger trace usually consumes more time to identify heap allocators in it.

When considering the feature A (i.e., returning a heap pointer) in Section 3.3.1, we could identify a set of candidate allocators, e.g., 43 allocators in Microsoft Word 2010. Further, after we apply other features, the set of candidate allocators becomes smaller. For example, after considering the group B features (i.e., use of allocation size), we only get 11 candidate allocators. Furthermore, only 5 candidate allocators are left after considering the group C features (i.e., use of the returned pointer).

We also did some manual analysis to validate the accuracy of these candidate allocators. Among the 5 candidate heap allocators in Microsoft Word, we figured 4 of them have a name indicating that they are allocators. After a further reverse engineering analysis, we confirmed that they are heap allocators. Only one extra function is wrongly identified as a heap allocator in this case.

Abstraction of Heap Access Operations. By recovering the high-level heap access operations, we could reduce the number of heap operation pairs. Table 5 shows some statistics of this abstraction. In addition to the trace information, this table also shows the number of allocation sites in the sample traces. In a trace, an allocator may be invoked several times, so there will be more allocation sites than heap allocator functions shown in Table 4.

On the right side of the table, it shows the number of heap access operations. Note that, for each heap access operation, its heap allocation site is unique. So the number of heap operation pairs equals to the heap access operations.

As shown in the table, there are too many low-level heap access instructions (i.e., type-D3 access in the table). The number of high-level heap access operations is much smaller. Furthermore, after we consider the taint attributes, the number of heap access operations drop quickly. Finally, we sorted the remaining heap access operations according to their type, taint attributes, constraint complexity etc. as discussed in Section 3.4.2. The number of heap operation pairs that are likely to be vulnerable is quite small, comparing to the number of low-level heap operations.

5.7 Comparison with fuzzers

In order to evaluate the effectiveness of HOTracer, we performed extra experiments, to compare it with existing vulnerability discovery solutions, especially fuzzing. As our prototype worked in Windows 7, we chose two representative fuzzers on Windows, i.e., WinAFL [3] and Radamsa [4], to test vulnerable softwares with the same seed inputs.

WinAFL is a fork of AFL on Windows, which relies on dynamic instrumentation using DynamoRIO [5] to measure and extract target coverage. Radamsa is a black-box fuzzer not guided by code coverage. Instead, it aims at testing execution path thoroughly, similar to our solution.

Due to the time limitation, we tested all these solutions on one application `potplayer`, with same seed inputs, for one day. WinAFL found no crashes during this time period, while Radamsa found 1144 crashes related to heap overflows.

With a further analysis, we figured out there are only 11 crash points, and 3 vulnerability points. HOTracer found all these 3 vulnerabilities, and 4 more heap overflows.

It is worth noting that, fuzzers and HOTracer both have other advantages. For example, general-purpose fuzzers

[2] http://samples.libav.org/

[3] https://github.com/ivanfratric/winafl

[4] https://github.com/aoh/radamsa

Table 4: Accuracy of the heap allocation recognition, including the statistics of the trace and the time of the heap allocation identifications. Type-A allocators are ones that satisfy the heuristics A in Section 3.3.1, i.e., functions return heap poitners. Type-A-B allocators are ones that satisfy both heuristics A and one of B1/B2/B3/B4. Similarly, type-A-B-C allocators satisfy one of C1/C2/C3/C4 in addition to previous heuristics. Confirmed allocators are ones that we manually validated to be real allocators, either from the symbol information of those functions, or from manual reverse engineering.

App.	Trace Info					Heap Allocations				
	Record Time	Snapshot Size	Record Size	Replay Time	Trace Size	Identif. Time	type-A A	type A-B	type A-B-C	confirmed allocators
demo	60s	632.5M	4.5M	444s	6.1M	1s	1	1	1	1
XnView	29s	430.6M	36.6M	486s	2.8G	125s	30	7	5	3
ZipItFast	83s	819.3M	13.6M	1257s	31.9G	4044s	28	9	3	3
MS Word	178s	539.8M	120.7M	3712s	35.9G	305s	43	11	5	4
Potplayer	131s	523.4M	40.6M	1676s	50.8G	714s	33	9	2	2
QQPlayer	150s	667.1M	138.2M	3320s	47.0G	630s	39	12	3	3
MPlayer	183s	519.7M	38.2M	1561s	16.6G	518s	22	4	2	2

Table 5: Abstraction of heap access operations. Type-D3 access operations are all low-level heap access instructions, as discussed in Section 3.4.1. Type-D2 access operations are `REP`-prefixed instructions or a short sequence of continuous heap access instructions. Type-D1 access operations are loops performing a single heap access.

App.	Trace Info		Alloc. Sites		Heap Access					
	Trace Size	Analy. Time	Alloc. Sites	Tainted Alloc.	type-D3 Access	type-D2 REP	type-D2 seq	type-D1 Access	Tainted Access	Sorted Pair
MS Word	35.9G	1097s	2,643	20	20,244,088	81,349	1,114,499	619,380	3,450	125
MS Word	535.6G	16210s	3,886	33	267,831,917	710,569	12,751,174	10 M.	29,718	1,258
Potplayer	21.9G	1231s	23,099	4,695	6,832,296	38,802	1,198,956	239,809	19,933	322
Potplayer	32.5G	695s	16,127	105	2,249,059	20,145	354,476	203,445	674	201
Potplayer	50.8G	2768s	18,773	154	2,061,109	14,435	405,334	130,501	1,078	254
Potplayer	63.9G	1267s	73,533	45	20,282,901	61,412	3,944,539	510,715	1070	109
Potplayer	106.1G	4312s	47,118	4,820	4,080,172	137,771	5,927,258	1 M.	27,466	630
QQPlayer	47.0G	2673s	28,749	10	6,488,402	141,384	1,375,956	910,115	12	10

could find other types of vulnerabilities, not only heap overflows. Our solution HOTracer could be used to triage the root cause of crashes, and help debugging and fixing bugs which are time-consuming and important to vendors.

5.8 Case studies

In this section, we will study some vulnerabilities in details and show some findings during the analysis.

5.8.1 Tainted Access Offset

As discussed in the background, we merged the access offset with the access size. In other words, the value of access offset is included in the access size. And whenever the access offset is tainted, we mark the access size as tainted. In the experiment, we found a new vulnerability in `Feiq` due to a tainted heap access offset.

5.8.2 Implicit Taint

Sometimes, the input does not directly affect the allocation size or access size. Instead, the sizes are control-dependent on the inputs.

A common case is that, developers use `strlen` or custom loops to identify the length of an input string, and allocate buffers. The program will compare the input bytes against the special character `'\0'`, and increase the allocation size accordingly.

Another example is that, the vulnerability CVE-2014-1761 in Microsoft Word uses an access size that is controlled by the number of `lfolevel` fields in the input. The program will compare the input against `lfolevel`, and set the access size accordingly.

Traditional taint analysis will not cover this type of implicit data flow. However, HOTracer could detect them by performing a backward data flow analysis on the access size and allocation size. If we found the access size or allocation size is control-dependent on the inputs, then we could report a candidate vulnerability.

5.8.3 Mismatch Taint

Heap overflow vulnerabilities may exist if either the allocation size or the access size is tainted. For developers, it is easier to realize the existence of heap overflow and deploy sanity checks when only allocation size or access

size is tainted. However, it gets challenging when both the allocation size and access size are tainted.

The allocation size and access size may be related to different input bytes. In this case, it is prone to heap overflow. For example, the access size in vulnerability CVE-2014-1761 is related to `lfolevel`, but its allocation size is related to another field `listoverridecount`.

A more common case is that, the access size and allocation size are relevant to the same input bytes, but they mismatch due to several causes.

First, the allocation size may get smaller than expected if there are integer overflows, e.g., two new vulnerabilities we found in QQPlayer and PotPlayer. Second, the access site may be so far from the allocation site that developers forget or fail to sanitize the inputs properly, e.g., the vulnerability CVE-2011-5233 in `InfraView`. Finally, the access size and allocation size may change due to dynamic features of programming languages. For example, in the vulnerability CVE-2015-0327 in Adobe Flash, the allocation and access size are both relevant to the API `nextNameIndex`. However, this API could be overridden by users, to cause mismatches and trigger heap overflow.

5.8.4 Multiple Vulnerabilities in One Trace

In some cases, there may be multiple potential heap vulnerabilities in a program path. For example, when analyzing the trace generated for the known vulnerability CVE-2014-1761 in Microsoft Word, we found several potential vulnerable points, and confirmed two of them are vulnerable. We also confirmed that these two new bugs still exist in the latest Microsoft Word (i.e., Office 2016). It shows that HOTracer could find out all potential heap vulnerabilities in one path, while existing solutions only focus on the first vulnerability.

5.8.5 Long Testing Time

Sometimes, the bugs will only be triggered after the program has run for a while. For example, a new we found in VLC could only be triggered after we play a video file for several minutes. Existing solutions like AFL could hardly find this type of vulnerabilities, because the throughput is extremely low. HOTracer is better at handling this kind of issues. It first filters seed inputs that could lead to different paths, and then analyzes each path once, no matter how long that path takes.

6 Related Work

6.1 Heap Overflow Detection

6.1.1 Static Analysis

Static analysis could be used to analyze programs without executing them. For example, Allamigeon et al. utilizes abstract intepretation to ensure the absence of heap overflow [2]. Chen et al. proposes a solution based on FSM (finite state machine) to report potential heap overflows [11]. SIFT is a static analysis system to generate input filters that nullify integer overflow errors associated with critical program sites such as memory allocation or block copy sites [27].

However, static analysis usually requires access to source code. And it is challenging to predict the spatial attributes of heap objects with static analysis, and thus hard to find heap overflows with static analysis. They will usually introduce a high false positives and false negatives. Moreover, static analysis solutions in general require a precise reachability and alias analysis, limiting their scope of use.

6.1.2 Online Dynamic Analysis

Dynamic analysis is more efficient to detect heap vulnerabilities, since it could get a precise spatial attributes and point-to relationship at runtime. There are also two types of dynamic analysis solutions: online analysis and offline analysis. Online dynamic analysis solutions usually first instrument target applications with metadata before execution, and then track the metadata and check security violations during the execution.

Online detection: AddressSanitizer [40] is one of the most effective solutions to detect heap (and other) vulnerabilities at runtime. It instruments redzones to each heap object when the object is allocated, and marks redzones' bytes as unaddressable while objects' bytes are addressable. A heap overflow (or underflow) vulnerability is reported if an unaddressable byte is accessed. This solution introduces a high runtime performance overhead (e.g., 73%), and is not suitable for production use. Moreover, it could only detect vulnerabilities when the given or generated input testcases could trigger security violations.

SoftBound [30] tracks the size and base of every pointer, and checks each pointer dereference operation. BaggyBounds [1] uses a compact table to store object sizes, adopts a fast algorithm to get object sizes and base addresses from only pointers, and checks each pointer arithmetic operation. Duck et al. tries to protect heap bounds with low fat points [20]. Diehard [4] and Dieharder [34] randomly allocate memory larger than required, and thus mitigate heap overflow vulnerabilities.

Online detection solutions rely on inputs to trigger vulnerabilities and help finding vulnerabilities in a passive way. Also they have reasonable high performance overheads.

Fuzzing: Fuzzing is another type of state-of-art solutions to detect vulnerabilities. Among them, AFL [51] is one of the most popular fuzzers. TaintScope [49] is a checksum-aware fuzzing tool which can identify checksum-based checks and bypass such checks. SYMFUZZ [10] combines both black- and white-box techniques to maximize the effectiveness of fuzzing. Zhiqiang

et al. utilizes the results of static analysis, and filter out sensitive input bytes using data lineage analysis [25]. Based on the analysis results, the fuzzer could only mutate target bytes to increase the efficiency. Driller [44] and VUzzer [38] are the most recent works on fuzzing. Driller enables AFL to explore new paths in an alternative way of fuzzing and concolic execution. And VUzzer enhances the efficiency of general-purpose fuzzers with a *smart* mutation feedback loop based on applications' control- and data-flow features.

Like other online detection solutions, fuzzers also rely on input testcases to trigger vulnerabilities at runtime. Moreover, they simply rely on program crashes to detect vulnerabilities, due to the lack of runtime metadata support. So they may not find vulnerabilities with strict conditions even if they have reached a very high code coverage. As they are general fuzzings tools and provide few supports for triaging crashes, it requires many manual efforts when further identifying root causes of found crashes.

Symbolic execution: Symbolic execution is a well-known technique used to reason applications. By marking inputs as symbolic values and propagateing them to variables, it could be used to analyze all possible states of one program path with only one-time analysis. Featuring with path exploration and vulnerability condition modeling, symbolic execution could be used to discover vulnerabilities. Although they are successful in many cases [7, 9, 14, 37], symbolic execution is rarely adopted in practice due to the limitations of complex constraint solving and path explosion. Traditional symbolic execution solutions mainly focus on how to explore new program paths and reduce the complexity of constraints.

Concolic execution [22, 29] is an alternative way for full symbolic execution. With concrete values, the analysis engine could explore deeper and be more scalable. Our solution adopts the similar offline trace-based constaint generation. However, we concentrate only on heap overflow vulnerabilities, and thus apply some optimizations to the symbolic execution and constraint solving process. As symbolic execution on real world applications is an open challenge, our solution does not improve it much, but roughly use it as a tool to reason about the constraints that we built with delicate data flow analysis.

6.1.3 Offline Dynamic Analysis

Offline dynamic analysis solutions usually analyze the runtime execution's results offline, and do not interfere the runtime execution except for recording. Comparing to online dynamic analysis, this type of solutions could perform in-depth analysis for a single dynamic execution, and explore potential vulnerabilities.

DIODE [41] targets heap allocation sites in a trace, and extracts and solves integer overflow conditions for allocation sizes to discover potential IO2BO (a special kind of heap overflow) vulnerabilities. It only considers heap allocation operations, but not heap access operations, and thus will miss many heap vulnerabilities. Moreover, it only considers integer overflow conditions, which is only a subset of heap overflow conditions.

Dowser [23] is an offline solution to detect buffer overflow (including heap overflow). It relies on compile-time information to filter pointer accesses in loops that are more likely to be vulnerable to buffer overflow (including heap overflow). It then uses dynamic taint analysis to infer which input bytes will affect these operations, and steers symbolic execution engine to explore the value space of the relevant input bytes. However, it does not support binary programs, and does not support precise heap layout analysis and thus are not efficient to find heap overflows. Moreover, it only considers heap access operations, but not heap allocation operations, and thus could not find all heap vulnerabilities. BORG [31] is the binary version of Dowser, facing a same set of limitations.

6.2 Related Program Analysis Techniques

MemBrush [13] proposes several heuristics to identify custom memory allocators. The key observation is that a malloc-like routine will return a heap address, and its client will use this return value to access memory. It uses dynamic testing to repeatedly validate candidate functions against the expected behaviour, to filter out real allocators. This solution could identify custom heap allocators more accurately. However, it could not be integrated into our offline analysis solution. We leave it as a future work.

Aligot [8] proposes a solution to identify loops in execution traces, and uses it to identify cryptographic functions in obfuscated binary programs. A recent paper [50] improves Aligot in identifying loop bodies. Jordi Tubella et.al. also proposed a solution [45] to identify loops dynamically. These solutions could handle more complicated loops than our solution. But they are over-qualified for our target, i.e., identifying loops used for heap access operations.

Recognizing structures in binary is also helpful for our work. HOTracer can benefit from related works [26, 42, 48] to identify elements inside objects. These could make HOTracer able to detect and discover sub-object overflow vulnerabilities.

7 Discussion

It is challenging to recognize all custom heap management functions, especially when analyzing the trace directly. Although the heuristics-based solution we took is not perfect, it indeed helps us find more heap vulnerabilities than state-of-art solutions. But our solution could definitely benefit from an improved recognition algorithm.

It is also challenging to abstract all heap access operations. Complicated access operations could be missed, and the access size and other attributes of these operations are hard to retrieve, making our prototype miss potential vulnerabilities. Related work (e.g., CryptoHunt [50]) on program semantics comprehension could help HOTracer, e.g., to handle more complex loops.

Some heap vulnerabilities may not crash target programs even if they are triggered. Our solution could find out this type of vulnerabilities. However, they could still be exploited in some cases. It would be interesting to assess whether these vulnerabilities are exploitable. It is one of our ongoing research to automatically analyze them.

Moreover, it is an open challenge to solve constraints. Vulnerability conditions and path constraints generated by HOTracer may be too complex to solve. In that case, we make efforts to make it practical and may still miss some potential heap vulnerabilities. Also there are some more complex situations (e.g., checksum mentioned in TaintScope [49], blocking checks mentioned in DIODE [41]) making it harder. In the evaluation we performed, we did not have this type of problems. But in general, it needs to be addressed. We could utilize the vulnerability conditions and candidate pairs of heap allocation and heap access operations, to perform other types of analysis, e.g., fuzzing, or change the path carefully by flipping like DIODE.

8 Conclusion

Heap overflows account for a big portion of real world memory corruption based exploits. We pointed out the root causes of heap vulnerabilities, and proposed a new offline dynamic analysis solution to discover heap vulnerabilities in program execution traces. It is able to explore each program path in depth to find vulnerabilities that are hard to detect and prone to miss by existing solutions. We also proposed several optimizations, making our solution more practical. Our prototype tool HOTracer found 47 new vulnerabilities in 17 real world applications, showing that this solution is effective.

Acknowledgement

We would like to thank our shepherd Stelios Sidiroglou-Douskos, and the anonymous reviewers for their insightful comments. This research was supported in part by the National Natural Science Foundation of China (Grant No. 61572483, 61402125 and 61502469), and Young Elite Scientists Sponsorship Program by CAST (Grant No. 2016QNRC001).

References

[1] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Usenix Security Symposium*, 2009.

[2] Xavier Allamigeon and Charles Hymans. Static analysis by abstract interpretation: application to the detection of heap overflows. *Journal in Computer Virology*, 4(1):5–23, 2008.

[3] Arash Baratloo, Navjot Singh, and Timothy Tsai. Libsafe: Protecting critical elements of stacks. *White Paper http://www. research. avayalabs. com/project/libsafe*, 1999.

[4] Emery D Berger and Benjamin G Zorn. Diehard: probabilistic memory safety for unsafe languages. In *ACM SIGPLAN Notices*, volume 41, pages 158–168, 2006.

[5] Derek L. Bruening. Efficient, transparent and comprehensive runtime code manipulation. Technical report, 2004.

[6] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *ISSTA*, 2012.

[7] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, 2008.

[8] Joan Calvet, José M Fernandez, and Jean-Yves Marion. Aligot: cryptographic function identification in obfuscated binary programs. In *CCS*, 2012.

[9] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy*, 2012.

[10] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *IEEE Symposium on Security and Privacy*, 2015.

[11] Shuo Chen, Jun Xu, Zbigniew Kalbarczyk, and K Iyer. Security vulnerabilities: From analysis to detection and masking techniques. *Proceedings of the IEEE*, 94(2):407–418, 2006.

[12] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. StackArmor: Comprehensive Protection From Stack-based Memory Error Vulnerabilities for Binaries. In *NDSS*, 2015.

[13] Xi Chen, Asia Slowinska, and Herbert Bos. Who allocated my memory? detecting custom memory allocators in c binaries. In *WCRE*, pages 22–31. IEEE, 2013.

[14] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, 2011.

[15] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Usenix Security Symposium*, 1998.

[16] Mark Daniel, Jake Honoroff, and Charlie Miller. Engineering heap overflow exploits with javascript. *WOOT*, 8:1–6, 2008.

[17] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[18] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. Tappan zee (north) bridge: mining memory accesses for introspection. In *CCS*, 2013.

[19] Gregory J. Duck and Lorenzo Yap, Cavallaro. Stack Bounds Protection with Low Fat Pointers. In *NDSS*, 2017.

[20] Gregory J Duck and Roland HC Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 132–142. ACM, 2016.

[21] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. Statically detecting use after free on binary code. *Journal of Computer Virology and Hacking Techniques*, 10(3):211–217, 2014.

[22] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, January 2012.

[23] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Usenix Security Symposium*, 2013.

[24] Etoh Hiroaki and Yoda Kunikazu. ProPolice: Improved stack-smashing attack detection. *IPSJ SIG Notes*, pages 181–188, 2001.

[25] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Convicting exploitable software vulnerabilities: An efficient input provenance based approach. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, 2008.

[26] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *NDSS*, 2010.

[27] Fan Long, Stelios Sidiroglou-Douskos, Deokhwan Kim, and Martin Rinard. Sound input filter generation for integer overflow errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 439–452, New York, NY, USA, 2014. ACM.

[28] Microsoft. Software vulnerability exploitation trends: Exploring the impact of software mitigations on patterns of vulnerability exploitation (2013). `http://download.microsoft.com/download/F/D/F/FDFBE532-91F2-4216-9916-2620967CEAF4/Software%20Vulnerability%20Exploitation%20Trends.pdf`.

[29] David Molnar, Xue Cong Li, and David A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, 2009.

[30] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *PLDI*, 2009.

[31] Matthias Neugschwandtner, Paolo Milani Comparetti, Istvan Haller, and Herbert Bos. The borg: Nanoprobing binaries for buffer overreads. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, 2015.

[32] Meining Nie, Purui Su, Qi Li, Zhi Wang, Lingyun Ying, Jinlong Hu, and Dengguo Feng. Xede: Practical Exploit Early Detection. In *RAID*, 2015.

[33] Nick Nikiforakis, Frank Piessens, and Wouter Joosen. Heapsentry: Kernel-assisted protection against heap overflows. In *DIMVA*, 2013.

[34] Gene Novark and Emery D Berger. Dieharder: securing the heap. In *CCS*, pages 573–584. ACM, 2010.

[35] PaX-Team. PaX ASLR (Address Space Layout Randomization). `http://pax.grsecurity.net/docs/aslr.txt`, 2003.

[36] Hendrik Post and Wolfgang Küchlin. Integrated static analysis for linux device driver verification. In *Integrated Formal Methods*, pages 518–537. Springer, 2007.

[37] David A. Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *Usenix Security Symposium*, 2015.

[38] Sanjay Rawat, Vivek Jain, Ashish Kumar, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS*, 2017.

[39] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy*, 2010.

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

[41] Stelios Sidiroglou-Douskos, Eric Lahtinen, Nathan Rittenhouse, Paolo Piselli, Fan Long, Deokhwan Kim, and Martin Rinard. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. In *ASPLOS*, 2015.

[42] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS*, 2011.

[43] Alexander Sotirov. Heap feng shui in javascript. *Black Hat Europe*, 2007.

[44] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, 2016.

[45] J. Tubella and A. Gonzalez. Control speculation in multithreaded processors through dynamic loop detection. In *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*, 1998.

[46] Arjan van de Ven and Ingo Molnar. Exec Shield. `https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf`, 2004.

[47] Vendicator. A "stack smashing" technique protection tool for Linux. `http://www.angelfire.com/sk/stackshield/`, 2000.

[48] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making Reassembly Great Again. In *NDSS*, 2017.

[49] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy*, 2010.

[50] Dongpeng Xu, Jiang Ming, and Dinghao Wu. Cryptographic Function Detection in Obfuscated Binaries via Bit-precise Symbolic Loop Mapping. In *IEEE Symposium on Security and Privacy*, 2017.

[51] Michal Zalewski. American fuzzy lop.

[52] Qiang Zeng, Mingyi Zhao, and Peng Liu. Heaptherapy: An efficient end-to-end solution against heap buffer overflows. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 485–496. IEEE, 2015.

[53] Chao Zhang, Tielei Wang, Tao Wei, Yu Chen, and Wei Zou. Intpatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In *Computer Security–ESORICS 2010*, pages 71–86. 2010.