Revery: from Proof-of-Concept to Exploitable
(One Step towards Automatic Exploit Generation)

Yan Wang[1,3], Chao Zhang[2*], Xiaobo Xiang[1,3], Zixuan Zhao[1,3], Wenjie Li[1,3], Xiaorui Gong[1,3*], Bingchang Liu[1,3], Kaixiang Chen[2],Wei Zou[1,3]

[1]{CAS-KLONAT[§], BKLONSPT[†]}, Institute of Information Engineering, Chinese Academy of Sciences
[2]Institute for Network Sciences and Cyberspace, Tsinghua University
[3]School of Cyber Security, University of Chinese Academy of Sciences
{wangyan9077,xiangxiaobo,zhaozixuan,liwenjie,gongxiaorui,liubingchang,zouwei}@iie.ac.cn,chaoz@tsinghua.edu.cn,
ckx1025ckx@gmail.com

ABSTRACT

Automatic exploit generation is an open challenge. Existing solutions usually explore in depth the *crashing paths*, i.e., paths taken by proof-of-concept (PoC) inputs triggering vulnerabilities, and generate exploits when *exploitable states* are found along the paths. However, exploitable states do not always exist in crashing paths. Moreover, existing solutions heavily rely on symbolic execution and are not scalable in path exploration and exploit generation. In addition, few solutions could exploit heap-based vulnerabilities.

In this paper, we propose a new solution `Revery` to search for exploitable states in paths diverging from crashing paths, and generate control-flow hijacking exploits for heap-based vulnerabilities. It adopts three novel techniques: (1) a *layout-contributor digraph* to characterize a vulnerability's memory layout and its contributor instructions; (2) a *layout-oriented fuzzing* solution to explore diverging paths, which have similar memory layouts as the crashing paths, in order to search more exploitable states and generate corresponding *diverging inputs*; (3) a *control-flow stitching* solution to stitch crashing paths and diverging paths together, and synthesize *EXP inputs* able to trigger both vulnerabilities and exploitable states.

We have developed a prototype of `Revery` based on the binary analysis engine `angr` [31], and evaluated it on a set of 19 real world CTF (capture the flag) programs. Experiment results showed that it could generate exploits for 9 (47%) of them, and generate EXP inputs able to trigger exploitable states for another 5 (26%) of them.

CCS CONCEPTS

• **Security and privacy → Penetration testing**; *Software reverse engineering*;

KEYWORDS

exploit; vulnerability; taint analysis; fuzzing; symbolic execution

1 INTRODUCTION

Due to the success of automated vulnerability discovery solutions (e.g., fuzzing), more and more vulnerabilities are found in real world applications, together with proof-of-concept (PoC) inputs. For example, Google's OSS-Fuzz platform [28] adopts several state-of-the-art fuzzers to continuously test open source applications, and has found over 1000 bugs in 5 months [4]. As a result, more and more human resources are spent on assessing vulnerabilities, e.g., identifying root causes and fixing them. It thus calls for solutions to automatically assess the severity and priority of vulnerabilities.

Vulnerability assessment, especially exploitability assessment, is important for both defenders and attackers. Attackers could isolate exploitable vulnerabilities and write exploits to launch attacks. On the other hand, defenders could prioritize exploitable vulnerabilities to fix first, and allocate resources accordingly. Moreover, defenders could learn from the exploits to generate IDS (Intrusion Detection System) signatures, to block future attacks.

A straightforward way to assess a vulnerability is *analyzing the program state at the crashing point*, i.e., the instruction leading to program crashes or security violations, which could be caught by a sanitizer (e.g., AddressSanitizer [29]). For example, Microsoft's `!exploitable` tool [3] inspects all instructions in the crashing point's basic block, and searches for known exploitable patterns, e.g., control transfer instructions with tainted targets. `HCSIFTER` [17] takes an extra step to recover the data corrupted by heap overflow, enabling the program to execute more code after the crashing point, and thus provides more reliable assessments. However, these solutions rely on heuristics to determine the exploitability of vulnerabilities, and thus are inaccurate sometimes. Moreover, they could not provide exploit inputs to prove the exploitability.

The ultimate way to assess the exploitability of a vulnerability is *generating a working exploit*, either by human or by machine, e.g., as demonstrated in Cyber Grand Challenge (CGC [14]). Sean Heelan proposed a prototype [18] in his thesis, using dynamic analysis and symbolic execution to generate exploits for classic buffer overflow vulnerabilities. AEG [8] and Mayhem [13] provide end-to-end systems to discover vulnerabilities and automatically generate exploits when possible, for source code and binary respectively. Q [26] and CRAX [20] could generate exploits for binaries given PoC inputs.

* Corresponding Authors.
§ Key Laboratory of Network Assessment Technology, CAS.
† Beijing Key Laboratory of Network Security and Protection Technology.

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10.
https://doi.org/10.1145/3243734.3243847

These automatic exploit generation (AEG) solutions share a similar workflow. In general, they will first analyze vulnerabilities in detail in the crashing paths using dynamic analysis, then search for exploitable states in *crashing paths*, then utilize symbolic execution to collect the path reachability constraints, vulnerability triggering constraints and exploit construction constraints respectively, and finally solve these constraints using SMT solvers and generate exploit inputs. However, these solutions could only solve a small number of problems. For example, machines developed in CGC could only solve 26 out of 82 challenge programs in the Final Event.

There are several challenges need to be addressed:

Challenge 1: Exploit derivability issue. As pointed in [15, 36], once memory corruption vulnerabilities are triggered, the victim program's state machine turns into a *weird (state) machine*. Exploitation is actually a process of programming the *weird machine* to perform unintended behavior. It is extremely important to set up the initial state of this weird machine in order to exploit it.

However, PoC inputs could corrupt some data and lead weird machines to non-exploitable initial states. For example, the program may exit soon after the crashing point due to some sanity checks. So, AEG solutions have to search for exploitable states not only in *crashing paths* taken by PoC inputs, but also in alternative *diverging paths*. This is known as *exploit derivability*, one of the core challenges of exploitation [36]. Few AEG solutions have paid attentions to this issue.

Challenge 2: Symbolic execution bottleneck. Existing solutions heavily rely on symbolic execution to explore program paths (e.g., for vulnerability discovery), or perform reasoning (e.g., for test case and exploit generation). AEG [8] and Mayhem [13] utilize symbolic execution to explore paths reachable from the vulnerability point and search for exploitable states, able to mitigate the aforementioned exploit derivability issue. However, symbolic execution has scalability issues and performs poorly in exploit generation.

First, it faces the *path explosion* issue when exploring paths, and consumes too many resources even when analyzing only one path. Second, it gets blind to certain exploitable states after concretizing some values. For example, it has to concretize symbolic arguments of memory allocations and symbolic indexes of memory access operations in a path, in order to model the memory states and enable exploring following sub-paths. But the concretized values could lead to non-exploitable memory states.

Challenge 3: Exploiting Heap-based Vulnerability. Few existing solutions could generate exploits for heap-based vulnerabilities. First, heap management functions are too complicated for program analysis techniques to analyze. For example, the single file `malloc.c` in the library `glibc` has more than 5000 lines of code. Second, heap management functions have deployed several sanity checks, which could detect the heap corruption at certain checkpoints.

Our solution. In this paper, we focus on the exploitability assessment of heap-based vulnerabilities, given PoC inputs. We present a framework `Revery`, able to search for exploitable states in not only crashing paths but also diverging paths and generate working control-flow hijacking exploits when possible.

First, it analyzes the vulnerabilities using dynamic analysis. Similar to existing AEG solutions, `Revery` also collects some runtime information in the crashing path, including taint attributes of variables. In addition, it inspects corrupted memory objects (denoted as *exceptional objects*), and objects that can be used to locate the exceptional objects. Moreover, it retrieves *layout-contributor* instructions from the path, which create these objects and set up the point-to relationship among them. Based on these instructions and objects, `Revery` creates a *layout-contributor digraph* to characterize the vulnerability's memory state and contributors.

Then it searches alternative diverging paths for exploitable states. `Revery` utilizes a novel *layout-oriented fuzzing* solution rather than symbolic execution to explore diverging paths. Similar to directed fuzzing [10], `Revery` also drives a fuzzer to explore paths close to specific targets, i.e., the crashing paths.

However, `Revery` does not aim at matching the exact crashing path or triggering the vulnerability during fuzzing. Instead, it ignores most of the instructions in the crashing path, but aims at hitting the aforementioned layout-contributor instructions, which could yield a similar memory layout as the vulnerability. Therefore, the fuzzer could explore many diverging paths, and has a better chance to find exploitable states, while sticking around the vulnerable memory states.

Finally, `Revery` tries to synthesize new *EXP inputs* to trigger both the exploitable states in diverging paths and vulnerabilities in crashing paths. It employs a novel *control-flow stitching* solution to stitch the diverging paths and crashing paths together, and then utilizes a *lightweight symbolic execution* to generate EXP inputs.

In certain cases, `Revery` is able to directly generate working exploits. But it is not guaranteed, due to the presence of complicated defense mechanisms, or the requirement of making dynamic decisions during exploitation, or other challenges which are out of the scope of this paper. It is worth noting that, even in cases where `Revery` fails to generate working exploits, `Revery` could provide *EXP inputs* to experts and help them write exploits.

Results. We have built a prototype of `Revery` based on the binary analysis engine angr [31], and evaluated it on 19 real world CTF (Capture The Flag) programs. It demonstrated that `Revery` is effective in triggering exploitable states, and could generate working exploits for a big portion of them. More specifically, `Revery` could generate exploits for 9 (47%) out of 19 programs, while existing open source AEG solutions could not solve any of them. Furthermore, it could trigger exploitable states for another 5 (26%) of them.

In summary, we have made the following contributions:

- We proposed an automated solution `Revery` able to transfer PoC inputs into EXP inputs, which could trigger vulnerabilities and enter exploitable states. It could also directly generate working exploits in certain cases.
- We proposed a *layout-contributor digraph* data structure, to characterize vulnerabilities' memory layouts and their contributor instructions, enabling many exploit-related analysis.
- We proposed a novel *layout-oriented fuzzing* solution, to search for exploitable states in diverging paths, without symbolic execution.
- We proposed a novel *control-flow stitching* solution, to stitch crashing paths and diverging paths together and synthesize *EXP inputs* with a lightweight symbolic execution.
- We have implemented a prototype of `Revery`, and demonstrated its effectiveness in real world CTF programs.

```
1.  struct Type1 { char[8] data;                                  };
2.  struct Type2 { int status;    int* ptr;  void init(){…};  };
3.  int (*handler)(const int*) = …;
4.  struct{Type1* obj1; Type* obj2;} gvar = {};
5.  int foo(){
6.    gvar.obj1 = new Type1;
7.    gvar.obj2 = new Type2;
8.    gvar.obj2->init();  // resulting different statuses
9.    if(vul)
10.     scanf("%s", &gvar.obj1->data);     // vulnerability point
11.   if(gvar.obj2->status)                // stitching point
12.     res = *gvar.obj2->ptr;             // crashing point
13.   else                                 // stitching point
14.     *gvar.obj2->ptr = read_int();      // exploitable point
15.   handler(gvar.obj2->ptr);             // hijacking point
16.   return res;
17. }
```

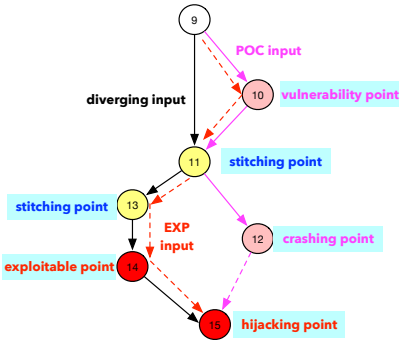


Figure 1: An example heap overflow. The vulnerability at line 10 could overwrite the following object, i.e., obj2. PoC inputs would crash at line 12 and enter a non-exploitable state. Successful exploits will trigger the exploitable state at line 14.

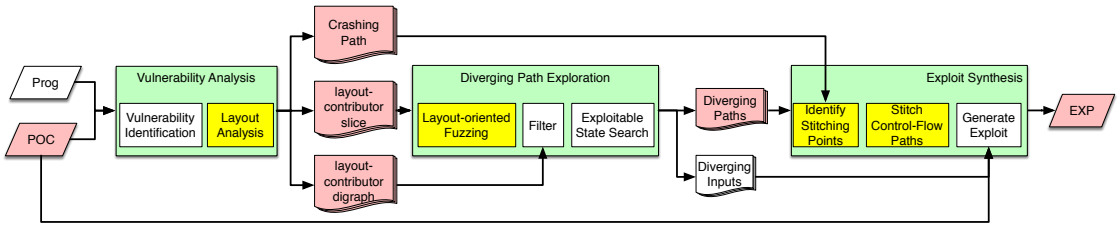


Figure 2: Overview of Revery. It first analyzes the vulnerability in the crashing path and gets the layout-contributor digraph to characterize the vulnerability, then guides a fuzzer with this digraph to explore diverging paths and search for exploitable states, and finally stitches the diverging path with the crashing path to synthesize exploits.

2 MOTIVATION EXAMPLE

In this section, we will illustrate the exploit derivability issue facing by automated exploit generation solutions, and present the overview of our solution `Revery`, with a running example demonstrated in Figure 1.

2.1 The Vulnerability

As shown in Figure 1, there is a heap overflow vulnerability at line 10. The two objects `obj1` and `obj2` have the same size, and are likely to be allocated next to each other in the heap. If the vulnerability condition `vul` at line 9 is met, lengthy inputs could cause an overflow in the buffer `obj1->data`. As a result, objects (e.g., `obj2`) following this buffer will be corrupted.

Therefore, the statement at line 12 and 14 will read from and write to corrupted memory address respectively. If the corrupted pointer `obj2->ptr` points to invalid (e.g., nonexistent) memory, these two statements will cause crashes. If it points to valid memory, the statement at line 12 will execute normally (but result in wrong return value), while the statement at line 14 will further corrupt the target memory and cause Arbitrary Address Write (AAW).

From the perspective of exploitation, the statement at line 12 is non-exploitable, unless the returned value *res* affects control-flow in caller functions. But the statement at line 14 triggers an exploitable state. It causes an AAW primitive able to overwrite arbitrary targets, including the global function pointer `handler` which is invoked at line 15, and thus could cause control-flow hijacking at line 15.

2.2 Exploit Derivability

As discussed in [36], exploit derivability is one of the core challenges of exploitation. More specifically, given a PoC input for a vulnerability, the program could be turned into a *weird machine*, but with a non-exploitable initial state. To successfully exploit the vulnerability, we have to search for exploitable states in alternative diverging paths, and lead the *weird machine* to exploitable.

As shown in the running example, assuming a PoC input proving the vulnerability at line 10 is provided (e.g., by fuzzers), it could overwrite the field `obj2->status` to non-zero, and overwrite `obj2->ptr` to invalid memory address, and cause a crash at line 12. So this PoC leads the *weird machine* to a non-exploitable initial state. A successful exploitation has to trigger the vulnerability (at line 10) and enter an exploitable state (e.g., at line 14).

For simplicity, we introduce several terminologies:

- **Crashing path:** the path taken by the PoC input, e.g., the path `9->10->11->12` in the example.
- **Crashing point:** the instruction where the program crashes or a security violation is caught by sanitizers, e.g., line 12.
- **Vulnerability point:** the instruction where the vulnerability (i.e., security violation) happens, e.g., line 10 in the example. A crashing path may have multiple security violations. The first violation point is denoted as the vulnerability point.
- **Exploitable point:** the instruction which could lead to a successful exploit, e.g., line 14 in the example. Exploitable points lead to exploitable states where the *weird machine* could work

properly. In practice, arbitrary address read/write/execute instructions (AAR/AAW/AAX) are classical exploitable points.
- **Diverging path:** the path where exploitable states could be found, e.g., `9->11->13->14` in the example.
- **Hijacking point:** the instruction where the control-flow could be hijacked, e.g., line 15 in the example. They are special exploitable points. In the running example, it is a second-order exploitable point, caused by the first exploitable point in line 14.
- **Exploitation path:** the path taken by a successful exploit, e.g., `9->10->11->13->14->15` in the example.
- **Stitching points:** special instructions in the diverging path and crashing path, which could be stitched together to generate the exploitation path, e.g., line 11 and line 13 in the example. In practice, there may be numerous sub-paths between two stitching points to explore.

It is worth noting that, the crashing point (line 12) in the running example could reach to the hijacking point (line 15), but it is not exploitable. As aforementioned, this hijacking point is a second-order exploitable point, made by the exploitable point in line 14. Without the help of line 14, line 15 could not be exploited.

So, to conduct successful exploitations, we have to think outside the box made by the original PoC, and search for exploitable states in diverging paths. This is the intuition of our solution and the origin of the name `Revery`. To the best of our knowledge, existing AEG solutions paid few attentions to this *exploit derivability* issue.

2.3 Our Solution: `Revery`

We proposed a novel solution `Revery`, to solve the *exploit derivability* issue and assess the exploitability of heap-based vulnerabilities. At the high level, `Revery` analyzes the vulnerability in detail, utilizes the vulnerability information to guide a fuzzer rather than symbolic execution to explore diverging paths and search for exploitable states, then synthesizes exploitation paths by stitching the crashing path and diverging path, and finally generates inputs to trigger both the vulnerability and exploitable states. As shown in Figure 2, it has three major components.

2.3.1 Vulnerability Analysis. `Revery` first analyzes the vulnerability in detail, similar to existing AEG solutions. It uses dynamic analysis to test target application with the provided PoC input. More specifically, it tracks the states of each pointer and memory object, and catches security violations along the crashing path. It could thus identify the vulnerability point, e.g., line 10 in Figure 1.

More importantly, it identifies *exceptional objects* corrupted by the vulnerability, e.g., `obj2` in the example. `Revery` also identifies the exceptional object's indexing objects, which could be used to locate the exceptional object, e.g., the global variable `gvar` in the example. Moreover, it retrieves *layout-contributor* instructions from the execution trace, which create the exceptional and indexing objects and set up their point-to relationships, e.g., line 7 in the example. These objects and contributor instructions are used to construct a layout-contributor digraph.

2.3.2 Diverging Path Exploration. `Revery` searches for exploitable states in diverging paths, to solve the *exploit derivability* issue. Rather than using symbolic execution, it employs fuzzing.

First, it employs a novel *layout-oriented fuzzing* solution to explore diverging paths. To facilitate exploit generation, only diverging paths with memory layouts similar as the PoC input's will be explored. So, it drives a fuzzer to explore paths close to the crashing path, in a similar way as directed fuzzing solutions [10]. But instead of using the full crashing path, it uses the aforementioned *layout-contributor* instructions as the fuzzer's guidance. The fuzzer could thus produce diverging inputs to exercise the diverging paths (e.g., `9->11->13->14` in the figure) with proper memory layouts.

Then, `Revery` searches for exploitable states in the diverging paths. Several heuristics are used to identify exploitable states. For example, if a memory store operation's destination is controlled by the corrupted object, e.g., line 14, it is an exploitable state.

Furthermore, `Revery` also searches for hijacking points in these diverging paths. Hijacking points sometimes are not obvious. So `Revery` uses some heuristics to infer hijacking points. For example, line 15 in the figure is a second-order hijacking point, which could be enabled if line 14 overwrites the global function pointer.

2.3.3 PoC Stitching. Once an exploitable state (together with a diverging input) in a diverging path is found, `Revery` will try to synthesize a new input to trigger both the vulnerability and the exploitable state. In general, it first finds the stitching points in the crashing path (e.g., line 11) and in the diverging path (e.g., line 13), with some specific data flow analysis.

Then it utilizes a lightweight symbolic execution to explore potential sub-paths between these two stitching points (e.g., `11->13`), and stitch the crashing path with the diverging path to synthesize an exploitation path (e.g., `9->10->11->13->14->15`), and finally generate inputs to exercise the exploitation paths. Several optimizations are deployed to make the symbolic execution lightweight.

Therefore, `Revery` could produce *EXP inputs* able to trigger both vulnerabilities and exploitable states. It could help experts to quickly generate working exploits. In certain cases, `Revery` is able to directly generate exploits. For example, `Revery` could generate an exploit input to hijack the control flow, by utilizing the exploitable state at line 14 to overwrite the global function pointer `handler`.

Assumptions. We assume the victim program is deployed in regular modern operating systems, with default defenses enabled (e.g., DEP [7] or sanity checks in `glibc`). Except that ASLR [23] is disabled, since it requires an extra information disclosure vulnerability or exploit, which `Revery` currently does not support.

3 VULNERABILITY ANALYSIS

To exploit a vulnerability, it is necessary to locate the vulnerability point and the program state at that point. Furthermore, to solve the exploit derivability issue, exploitable states around the vulnerability state should be searched for. Therefore, `Revery` performs *vulnerability identification* to locate the vulnerability, and performs *layout analysis* to characterize the vulnerability state.

3.1 Vulnerability Identification

Given a PoC input, `Revery` first needs to identify its corresponding vulnerability point. Dozens of solutions have been proposed to detect memory errors, e.g., AddressSanitizer [29]. However, AddressSanitizer will slightly change the memory layout of target applications, and thus is not suitable for exploit generation.

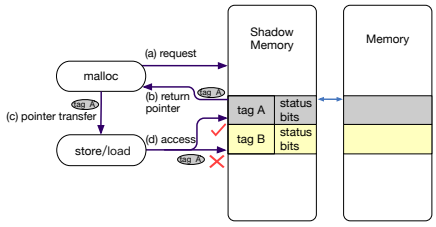
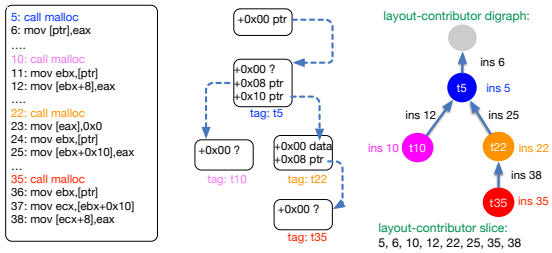


Figure 3: Illustration of heap-based vulnerability identification. Each heap object and pointer is associated with a memory tag. An extra status is attached to each memory object.



(a) sample execution trace (b) layout digraph (c) layout-contributor

Figure 4: An example layout-contributor digraph. Assume the object created at line 35 is an exceptional object. It could be indexed by objects created at line 22 and 5 repsectively, and eventually pointed by a global pointer `ptr`.

`Revery` utilizes a different technique, named memory tagging (MT, also known as memory coloring, memory tainting, lock and key) to locate vulnerabilities. A recent work [30] has implemented memory tagging in hardware. However, it encodes tags in memory pointers and thus affects the program states. Moreover, it only detects spatial memory violations, but not temporal violations.

`Revery` uses a shadow memory to non-intrusively track the tags of pointers and heap objects. It also tracks the status of heap objects, enabling detection of not only spatial vulnerabilities (e.g., heap overflow) but also temporal vulnerabilities (e.g., use-after-free).

In principle, each pointer is expected to access a specific memory object of valid status. If it is used at runtime to access an object of different tags or invalid status, then a security violation is caught. Figure 3 shows an example of vulnerability identification.

3.1.1 Memory tags. Each heap object and pointer is attached with a memory tag, indicating its lineage. This tag will be uniquely generated when an object is created, and propagate to the object's pointers and other related pointers (similar to taint analysis). Moreover, each heap object is associated with a status, i.e., `uninitialized`, `busy`, or `free`, standing for three status in its life-cycle, i.e., allocated but not initialized, initialized, or freed. It is worth noting that, a freed memory region could be allocated to new objects, and its memory status and tag will change accordingly.

In some corner cases, developers could use one object's pointer to get another object's pointer, with an arithmetic operation. It will wrongly propagate the first object's tag to the second pointer. Fortunately, this is rare for heap objects, since the offsets between heap objects are not fixed. The only exception is heap management functions, which could inspect adjacent objects in this way, no matter what semantics these objects would have. So `Revery` will disable tag propagation and validation for these special functions.

3.1.2 Security rules. For each heap memory access instruction (i.e., load and store), we could get the pointer's tag `tag_ptr` and target memory region's tag `tag_obj` and status `status_obj`. The memory access must not violate the following security rules:

- **V1: access intended objects**: Instructions should only access intended objects, i.e., `tag_obj` and `tag_ptr` must match.
- **V2: read busy objects**: Load instructions should not access freed or uninitialized memory, i.e., `status_obj` must be `busy`.
- **V3: write alive objects**: Store instruction should not access freed memory, i.e., `status_obj` must be `busy` or `uninitialized`.

Any violation of these rules will cause a vulnerability. For example, a buffer overflow memory access will violate rule V1. An uninitialized vulnerability will violate rule V2. A use-after-free (UAF) vulnerability could violate either V1, V2 or V3. If the freed object's memory has not been taken by other objects, then read access to it will violate V2, and write access to it will violate V3. If the freed object's memory is taken, then its tag will change, and any access to it via the original dangling pointer will violate V1.

3.2 Layout Analysis

`Revery` further analyzes object layouts to characterize the vulnerability state and retrieve instructions contributing to the state.

3.2.1 Vulnerability-related Object Layout. Each heap-based vulnerability (including heap overflow and UAF) is related to one *exceptional object*, whose content is (or will be) corrupted by the vulnerability. Further operations on these objects could lead the *weird machine* to exploitable states.

Assume the vulnerability point uses a pointer with tag `tag_ptr` to access a target object with tag `tag_obj`. If it is a write access, the object with tag `tag_obj` is the exceptional object, which will be corrupted by this write access. If it is a read access and this vulnerability is a UAF, the object with tag `tag_ptr` is the exceptional object, which will be corrupted by new object allocations that take the same memory. `Revery` currently does not support other types of read access violation well.

Further, `Revery` also tracks all indexing objects that can be used to locate exceptional objects. These exceptional objects and indexing objects are connected with the point-to relationship. As a result, `Revery` could get a digraph of objects, denoted as *layout digraph*. This layout digraph characterizes the vulnerability state to some extent. Figure 4(b) shows an example layout digraph.

3.2.2 Vulnerability-related Code. As aforementioned, the *weird machine* has to enter specific initial states, including the vulnerability state. So, it is necessary to prepare a similar object layout as the vulnerability's, both in diverging paths and exploitation paths. Thus, instructions contributing to the layouts are important.

We can see that, the following two types of operations could contribute the object layout: (1) memory allocations that creates new objects, and (2) store operations that assign an object's field with a pointer to another object. As a result, `Revery` could retrieve

all such contributor operations, which operate on objects in the layout digraph, and generate a *layout-contributor digraph*.

More specifically, each node in this digraph is an exceptional object or an indexing object, with an attribute of the object's creator instruction and memory tag. Each edge in the digraph represents a point-to relationship between two objects, with an attribute of the pointer assignment instruction. Given a target exceptional object, we could use backward slicing to construct this digraph. Figure 4(c) shows an example layout-contributor digraph. This digraph has a simpler form, called *layout-contributor slice*, which is a sequence of contributor instructions in execution order.

4 DIVERGING PATH EXPLORATION

To solve the exploit derivability issue, it is necessary to explore diverging paths and search exploitable states in them. In this section, we will introduce how `Revery` explores diverging paths.

4.1 Alternative Choices

Existing automated exploit generation solutions, e.g., AEG [8] and Mayhem [13], heavily rely on symbolic execution to explore the crashing path or reachable paths from the vulnerability point, in order to search exploitable states along the path exploration. However, symbolic execution has several severe challenges, and is not suitable for path exploration or exploitable state searching.

First, it is not scalable in path exploring. It suffers from the path explosion issue caused by branches and loops in programs. Even when analyzing one path, it costs too many resources. Moreover, the symbolic constraints are often too complicated to solve.

Second, symbolic execution may get blind to certain exploitable states. It has to concretize some symbolic values along the exploration, by adding extra constraints of concretized value assignments. It is impossible to try all candidate concretized values, thus misses certain values and causes blindness to certain exploitable states.

For example, it will concretize the symbolic arguments of *memory allocation* in a path, in order to model the memory states and explore following sub-paths. It is likely that only a small number of allocations could cause exploitable states. So the concretized memory allocation may lead to a non-exploitable state.

Moreover, it will also concretize *symbolic indexes* in memory access operations, because otherwise the operations' results are unknown. Similarly, it could also lead to non-exploitable states.

Another choice is exploring paths with the combination of fuzzing and symbolic execution, e.g., Driller [34]. However, the symbolic execution component of such solutions still have the aforementioned challenges. Moreover, the fuzzing component usually lacks of targets, and thus is not effective at finding exploitable states.

4.2 Layout-Oriented Fuzzing

`Revery` utilizes fuzzing solely to explore diverging paths and search for exploitable states. As shown in the field of vulnerability discovery, fuzzing is more effective than symbolic execution in exploring paths and program states. So, it is likely that fuzzing could also help find diverging paths and exploitable states faster.

`Revery` employs a novel layout-oriented fuzzing solution guided by the layout-contributor digraph, to explore diverging paths that build similar memory layouts as the vulnerability.

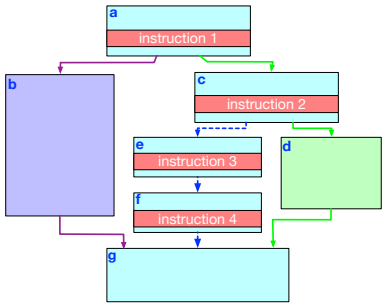


Figure 5: An example layout-contributor slice. Compared to the path a=>b, the path a=>c=>d has a longer prefix with the target slice in the path a=>c=>e=>f.

4.2.1 Design. `Revery` extends the popular coverage-guided fuzzer AFL to perform fuzzing. Instead of relying solely on code coverage to guide path exploration, `Revery` uses layout-contributor digraph as a guidance to tune the direction of exploration and mutation.

Similar to directed fuzzing [10], `Revery` drives the fuzzer to explore paths close to the crashing path. It only aims at matching instructions in the layout-contributor slice, and ignores other instructions in the crashing path. The design choices are made from the following three intuitions.

For simplicity, we introduce several terminologies. Given an input I_a, it could hit several layout-contributor instructions (maybe not in the same order as the guiding slice). The full list of such instructions is denoted as L_a, and its longest common subsequence (LCS) with the target guiding slice is denoted as P_a.

- **Intuition 1:** An input that hits all layout-contributor instructions, in the same order as the guiding slice, could construct a similar memory layout as the vulnerability.
  Layout-contributor instructions are responsible for creating the exceptional object of a vulnerability and its indexing objects, as well as setting the point-to relationships among them. So, an input hitting the full layout-contributor slice could probably construct similar memory layouts.
- **Intuition 2:** An input that hits a longer subsequence of the guiding slice is more likely to derive inputs hitting the full slice.
  In other words, if input I_a's LCS P_a is longer than I_b's LCS P_b, then the input I_a is better than I_b. As shown in Figure 5, assuming the target slice is in path a=>c=>e=>f, then an input exercising the path a=>c=>d is better than other inputs exercising a=>b. Further mutations on this input could derive inputs hitting the full guiding slice faster.
- **Intuition 3:** Inputs hitting fewer layout-contributor instructions are more likely to introduce fewer troubles for further exploit generation.
  In other words, for two inputs I_a and I_b, if their LCS P_a and P_b have a same length, but the layout-contributor instruction list L_a is longer than L_b, then the input I_b is better than I_a. In this case, the input I_a has more duplicated or out-of-order contributor instructions than I_b, which could cause redundant object creation or layout construction, making the memory layout too complicated to exploit.

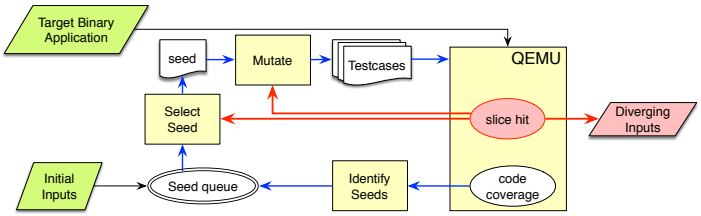


Figure 6: Illustration of the fuzzer implemented by `Revery`.

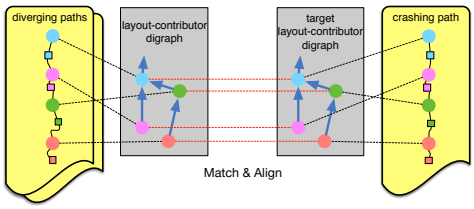


Figure 7: Filter diverging paths by matching their layouts against target crashing path's and aligning them when matched.

4.2.2 Implementation Details. `Revery` extends the popular fuzzer AFL [37]. As shown in Figure 6, AFL applies a continuous loop to explore paths. It (1) keeps a queue of good testcases, i.e., seeds; and (2) selects a seed from the queue; and then (3) mutates the seed to get a bunch of new testcases, and then (4) run the target binary program with the generated testcases in QEMU, and track the coverage, and then (5) identify seeds based on coverage information. `Revery` modifies AFL in the following two aspects.

Tracking Slice Hit Count. `Revery` adds an extra buffer `HIT` in the shared memory between QEMU and the fuzzer driver, together with the existing bitmap used for code coverage tracking. `HIT[0]` is used to track the count of slice hit, while `HIT[i]` is used to track whether the i-th instruction in the guiding slice has been hit or not.

More specifically, each time a layout-contributor instruction is executed, QEMU will increase the slice hit count `HIT[0]`. If this instruction is the n-th (n>=1) instruction in the guiding slice, then QEMU will set `HIT[n]` if and only if `HIT[n-1]` has been set. In this way, the fuzzer driver could get the slice hit count in `HIT[0]`, and the LCS of guiding slice in `HIT[1:N]`.

Tuning Fuzzing Directions. `Revery` modifies the fuzzer driver to make use of the collected slice hit information. Basically, it slightly changes the algorithms of seed selection. When picking up a seed from the queue to mutate, it first prioritizes seeds that have longer LCS, as discussed in Intuition 2. Then among seeds with LCS of same length, it prioritizes seeds with fewer slice hit count, as suggested in Intuition 3. Finally, it prioritizes seeds with smaller size and faster execution time, same as AFL's default policy.

4.3 Diverging Inputs Filtering

With layout-oriented fuzzing, `Revery` could find diverging inputs able to trigger the same layout-contributor slice as the PoC input. However, unlike layout-contributor digraph, the data flow constraints are missing in the layout-contributor slice. So the diverging inputs sometimes do not match the target layout-contributor digraph built from the crashing path. `Revery` thus takes an extra step to isolate diverging inputs that could match the target layout-contributor digraph.

In general, it first aligns the diverging path with the crashing path, and locates the instructions responsible for creating the exceptional object. Then, it constructs a new layout-contributor digraph of the exceptional object from the diverging path by backward slicing, in a same way as the crashing path. Finally, it matches this new digraph against the target digraph, by comparing each node's memory tag and its creator instruction's address in two digraphs. Figure 7 shows an example of how the match works.

If these two digraphs do not match, then this diverging input will be discarded. Otherwise, the diverging input is kept. Moreover, these two digraphs' nodes (i.e., heap objects) will be aligned accordingly, as well as the memory tags of all nodes. So, we could infer each object's counterpart between the diverging path and crashing path, enabling further common analysis on these two paths.

4.4 Exploitable States Searching

Even if the diverging paths have similar layouts as the vulnerability, not all of them are exploitable. `Revery` further removes diverging paths that do not have exploitable states.

4.4.1 Exploitable State. The exceptional object could affect other objects, and sometimes will be directly or indirectly used in some sensitive operations. The program states resulting from these sensitive operations are denoted as exploitable states.

In this paper, we mainly consider two types of sensitive (exploitable) operations, i.e., `memory write` and `indirect call`. For example, if the target address of a memory write is affected by the exceptional object, then attackers may control where to write and cause AAW (arbitrary address write), i.e., a commonly used exploitable state in practice. If attackers could affect the target of indirect calls, including virtual function calls and indirect *jmp* instructions etc., then they could hijack the control flow. In addition, `Revery` offers a template for experts to extend the definition of exploitable points, e.g., operations launching the unlinking attack.

4.4.2 Exploitable States Searching. This problem thus becomes identifying sensitive instructions whose operands are affected by the exceptional objects. Taint analysis is a straightforward solution.

`Revery` marks each object creation operation as a taint source, and attaches a unique taint label to it. Each operation propagates all source operands' taint labels to the destination. At each sensitive instruction (i.e., memory write or function call), the target address' taint labels will be checked if they contain the exceptional object's taint label. If yes, then this sensitive instruction is exploitable.

5 EXPLOIT SYNTHESIS

In this section, we will introduce how to synthesize new exploits from PoC inputs and diverging inputs.

Once an exploitable state is found in a path, existing AEG solutions usually generate exploits by solving the path, vulnerability and exploit constraints. However, as discussed in Section 4.1, symbolic execution solely is not effective in exploit generation.

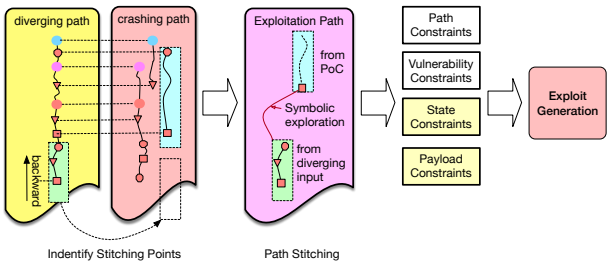


Figure 8: The workflow of exploit synthesis

Therefore, Revery uses symbolic execution as few as possible. It uses a lightweight symbolic execution as a bond to stitch the crashing path and diverging path together, and reuses the PoC input and diverging inputs to further reduce complicated constraints, making symbolic execution more practical.

Figure 8 shows the general workflow of exploit synthesis. In practical, it first identifies stitching points, and then explores sub-paths between stitching points and synthesize exploitation path, and finally solve related constraints to generate working exploits.

5.1 Identify Stitching Points

We first introduce how Revery identifies stitching points in both the crashing path and diverging path.

5.1.1 Stitching Points in the Crashing Path. In order to successfully exploit the victim program, its vulnerability must be first triggered, and some exceptional objects are corrupted. Revery thus chooses locations where exceptional objects are corrupted in the crashing path as stitching points.

As mentioned in Section 3.2.1, in the crashing path, each write access violation corrupts an exceptional object, and thus it is a candidate stitching point. For each read access violation in a UAF vulnerability, the exceptional object is the one that has been freed but still pointed by the dangling pointer. This exceptional object's memory region will be occupied by another memory allocation. Revery takes the new memory allocation operation as a candidate stitching point.

Since there could be multiple violations in one crashing path, there could also be multiple stitching points. Revery will try to stitch each of them with the diverging path.

5.1.2 Stitching Points in Diverging Paths. In order to successfully exploit the victim program, exploitable operations must be performed on corrupted exceptional objects or collateral objects.

What are good stitching points? Every instruction could be used as stitching points. But not all of them are good ones. A proper stitching point should satisfy several criterions:

- *Not too close to entry points.* Otherwise, many duplicated operations as the crashing path will be performed. Since duplicate operations (e.g., object initializations) will not happen in a legitimate control flow, it is infeasible to find a path to connect this stitching point with its counterpart in the crashing path.
- *Not too close to exploitable points.* Otherwise, a longer path is required to connect this stitching point with its counterpart, requiring more efforts of symbolic execution. The stitching point can be set before certain operations, e.g., initialization of exploitable points' operands, to save symbolic execution efforts.
- *Minimum data dependency.* The data flow after the stitching point in the diverging path should have few intersections with the data flow before the stitching point in the crashing path.

How to find stitch points? At a high level, Revery matches the diverging path's data dependency against the crashing path's, and locates the differences. Then it uses the instruction which causes the differences in the diverging path as stitching point.

First, Revery builds the layout-contributor digraph of the exploitable operation's operand in the diverging path. Then it matches this digraph against the digraph of the exceptional object in the crashing path. If the former is a sub-graph of the latter, it means the crashing path has already set up all data dependencies for the exploitable operation. Then, the instruction in the diverging path, which is right after the last write access to the exploitable operations' operands, is chosen as the stitching point.

Otherwise, there are different nodes or edges in the diverging path's digraph, i.e., the diverging path has alternated the dependency of the exploitable operations. In this case, Revery chooses the earliest instruction (object creation or write) in the diverging path, which causes differences in the digraph, as the stitching point.

5.2 Control-Flow Path Stitching

In order to stitch the crashing path and the diverging path together, Revery explores potential sub-paths connecting the stitching points in these paths. In general, it relies on symbolic execution to explore paths. However, Revery utilizes several heuristics to efficiently guide symbolic execution.

First of all, Revery uses the function call stack to guide the path exploration. It inspects the call stacks at the two stitching points respectively, and finds the differences. Figure 9 shows two example call stacks. These differences in call stacks indicate the direction of path exploration. Function invocations in the crashing path (e.g., `g1, g2, ..., gM` in the figure) should be returned one by one first, while function invocations in the diverging path (e.g., `h1, h2, ..., hK` in the figure) should be called one by one later.

In other words, when exploring potential paths, Revery will add the return instruction of function `gM, ..., g2, g1` as target instructions one by one, and then add the entry point of function `h1, h2, ..., hK` as target instructions one by one. These target instructions are dominator points between the two stitching points. Then Revery will explore potential sub-paths between these intermediate target instructions.

Revery further mitigates the sub-path exploration by reusing existing paths. For example, if there is already a sub-path connecting two intermediate destinations in either the diverging path or the crashing path, Revery will reuse this sub-path. Revery also performs a simple loop identification algorithm, and finds a sub-path to escape the loop as soon as possible, in order to reduce the burden of symbolic execution. Sometimes, the reused sub-path would cause the overall path unsolvable, Revery will try to remove these sub-paths and search for alternative sub-paths.

In this way, Revery greatly reduces the burden of symbolic execution when exploring sub-paths to connect the stitching points.

	Name	CTF	Vul Type	Crash Type	Violation	Final State	EXP. Gen.	Rex	GDB Exploitable
	woO2	TU CTF 2016	UAF	heap error	V1	EIP hijack	YES	NO	Exploitable
CONTROL FLOW HIJACK	woO2_fixed	TU CTF 2016	UAF	heap error	V1	EIP hijack	YES	NO	Exploitable
	shop 2	ASIS Final 2015	UAF	mem read	V1	EIP hijack	YES	NO	UNKNOWN
	main	RHme3 CTF 2017	UAF	mem read	V1	mem write	YES	NO	UNKNOWN
	babyheap	SECUINSIDE 2017	UAF	mem read	V1	mem write	YES	NO	UNKNOWN
	b00ks	ASIS Quals 2016	Off-by-one	no crash	V1	mem write	YES	NO	Failed
	marimo	Codegate 2018	Heap overflow	no crash	V1	mem write	YES	NO	Failed
	ezhp	Plaid CTF 2014	Heap overflow	no crash	V1	mem write	YES	NO	Failed
	note1	ZCTF 2016	Heap Overflow	no crash	V1	mem write	YES	NO	Failed
	note2	ZCTF 2016	Heap Overflow	no crash	V1	unlink attack	NO	NO	Failed
EXPLOIT-ABLE STATE	note3	ZCTF 2016	Heap Overflow	no crash	V1	unlink attack	NO	NO	Failed
	fb	AliCTF 2016	Heap Overflow	no crash	V1	unlink attack	NO	NO	Failed
	stkof	HITCON 2014	Heap Overflow	no crash	V1	unlink attack	NO	NO	Failed
	simple note	Tokyo Westerns 2017	Off-by-one	no crash	V1	unlink attack	NO	NO	Failed
	childheap	SECUINSIDE 2017	Double Free	heap error	V1	-	NO	NO	Exploitable
FAILED	CarMarket	ASIS Finals 2016	Off-by-one	no crash	V1	-	NO	NO	Failed
	SimpleMemoPad	CODEBLUE 2017	Heap Overflow	no crash	-	-	NO	NO	Failed
	LFA	34c3 2017	Heap Overflow	no crash	-	-	NO	NO	Failed
	Recurse	33c3 2016	UAF	no crash	-	-	NO	NO	Failed

Table 1: List of CTF pwn programs evaluated with `Revery`. Out of 19 applications, `Revery` could generate exploits for 9 of them, and generate EXP inputs to trigger exploitable state for another 5 of them, and failed for the rest 5.

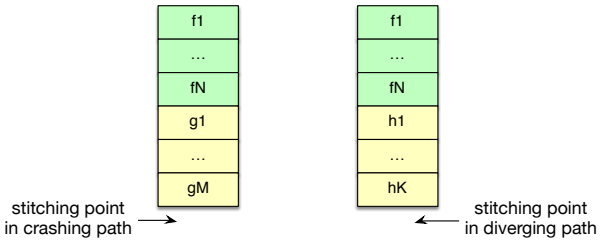


Figure 9: Example call stacks of stitching points

5.3 Exploit Generation

Once a sub-path connecting two stitching points is found, a candidate exploitation path is constructed. `Revery` could also solve the vulnerability constraints, path constraints and exploit constraints to generate final exploit samples. However, it is inadequate.

5.3.1 Exploitable State Constraints. Simply solving constraints of the exploitation path may not trigger the same exploitable state as the diverging path. `Revery` thus adds several extra data constraints to the exploitation path, ensuring the program state is still exploitable.

First, the memory allocation sizes in the exploitation path should be the same as the diverging path, in order to trigger the exploitable states as in the diverging path. `Revery` records the concrete sizes of all memory allocations when analyzing the diverging path. In the exploitation path, if a memory allocation which was in the diverging path has a symbolic size, then `Revery` will add a constraint to ensure this size equals to the concrete value in the diverging path.

Second, `Revery` will align the digraph of the crashing path with the diverging path's. Certain symbolic addresses in the diverging path are logically the same as their counterparts in the crashing path. So, in the stitched exploitation path, extra constraints must be introduced to claim the equality between these symbolic addresses.

5.3.2 Payload Constraints. With the aforementioned exploitable state constraints, together with the vulnerability and path constraints, `Revery` is able to generate *EXP inputs* to trigger both exploitable states and vulnerabilities. These inputs could help security experts to construct a full exploit.

In certain cases, `Revery` is able to directly generate working exploits. At the exploitable point, `Revery` could construct payload constraints which could lead to control flow hijacking. If the exploitable point is a function call (e.g., indirect *call* or `jmp` instruction) and its target is a symbolic value, `Revery` adds an extra constraint to set the target to attacker controlled value. If the exploitable state is a write access, and both the destination address and content to write are symbolic, then `Revery` adds an extra constraint to overwrite a known address (e.g., Global Offset Table entries or global function pointers) with attacker controlled value.

In this way, `Revery` could generate exploits to hijack control-flow for certain cases. However, it is not always guaranteed to succeed.

6 EVALUATION

We implemented a prototype of `Revery` based on the binary analysis engine angr [31] and the popular fuzzer AFL [37]. It consists of 1334 lines of code to analyze vulnerabilities, 190 lines of code to explore diverging paths with fuzzing, and 1249 lines of code to stitch paths and generate exploits.

In this section, we present the evaluation results of this system. The experiments are conducted in a Ubuntu 17.04 system running on a server with 115G RAM and Intel Xeon (R) CPU E5-2620 @ 2.40GHz*24. We evaluated `Revery` against 19 vulnerable programs

collected from 15 real world CTF (capture the flag) competition, 14 of them can be found in CTFTIME [1]. [1]

To thoroughly evaluate the effectiveness of Revery, we selected the target programs from CTF events based on the following rules: (1) no source code or debug symbols exist for these programs; (2) each program must have at least one heap-based vulnerability; (3) the diversity of vulnerability types must be large; and (4) the quality of the source CTF events is well acknowledged.

All programs are tested in a regular modern Linux operating system (Ubuntu 17.04), with the defense DEP [7] enabled. Unlike traditional environments, we disabled ASLR [23] in the evaluation. In practice, an information disclosure vulnerability or exploit is required to bypass ASLR. The current prototype of Revery could not generate information disclosure exploits yet.

6.1 Exploits by Revery

Table 1 shows the list of programs we evaluated. Out of 19 programs, Revery successfully exploited 9 of them, i.e., able to hijack their control flow. Revery could trigger the exploitable states for 5 more programs, i.e., providing exploit primitives for experts to launch successful exploits. It failed to analyze the rest 5 programs. More details will be discussed later.

This table also shows in detail the name and CTF event of each program. It shows the type of the known vulnerability in each program, including heap overflow, off-by-one, UAF and double free. Further, it shows the crash type of each vulnerability, i.e., results of applying PoC inputs to the vulnerable programs. Some of them are caught by the memory manager's sanity checks (denoted as `heap error` in the table), some others crash at invalid memory read instructions. Most of them do not even crash.

In addition, it shows the violation type of each vulnerability detected by Revery, the final exploitable state triggered by Revery, and whether Revery could generate exploits or not. Revery could detect security violations in 16 out of 19 programs. It could trigger exploitable states of EIP hijacking, arbitrary memory write, and unlink attack for 3, 6 and 5 programs respectively. Revery could generate working exploits for first two types of exploitable states.

As a comparison, we also evaluated the open-source AEG solution Rex [5] provided by the Shellphish team and the `exploitable` plugin in GDB on these programs. As shown in the last two columns of the table, Rex could not solve any of these programs, and GDB `exploitable` simply assesses the exploitability based on crash type.

6.2 Case Studies

In this section, we investigated these programs in detail, and analyzed why our solution Revery succeeded or failed.

6.2.1 **Control-Flow Hijacking Exploits.** Revery successfully generated control-flow hijacking exploits for 9 programs. With the given PoC inputs, 2 programs corrupt the heap metadata and are caught by the sanity checks deployed in `glibc` memory allocator. Three other programs crash at invalid memory read instructions, whose results are only dumped by functions like `printf`, which could not cause control-flow hijacking. The rest 4 programs do not even crash with the provided PoC.

Limit of State-of-the-art AEG Solutions. Such vulnerabilities are usually considered as non-exploitable by exploitability assessment tools. To successfully exploit these vulnerabilities, we have to avoid the metadata corruption being caught by sanity checks, and accurately model the memory allocator if using symbolic execution.

So state-of-the-art AEG solutions could not generate exploit automatically for them. We have tested all these programs with Rex[5], an automated exploit generation tool that developed by the Shellphish team, which won the first in offense in CGC. But it failed to generate exploits for any of them.

Performance of Revery. By exploring exploitable states in diverging paths, Revery can generate exploits for all 9 programs. For example, `WoO2` and `WoO_fixed` crash because one object is freed twice. To exploit this kind of vulnerabilities, heap Fengshui [32] is needed, which is too complicated for automated solutions. Instead, Revery goes back to the vulnerability point, and finds a diverging path which could lead to EIP hijack.

Three of the exploitable states could hijack the program counter, and the other six could cause arbitrary address write (AAW). AAW is a well-known exploit primitive, could enable many exploits. For example, it could be used to modify the global offset table (GOT) and hijack the control flow.

6.2.2 **Exploitable States.** Sometimes Revery is not able to generate working exploits, even if it has found the exploitable states and stitched an exploitation path. As shown in the table, Revery could trigger exploitable states but fail to generate working exploits for 5 programs.

For these programs, there is no critical data fields (e.g., function pointer, VTable pointer etc.) in the exceptional object, and it is extremely challenging to automatically generate exploits against them. Instead, we have to utilize the corrupted metadata in the exceptional objects to exploit the specific heap allocators.

Revery utilizes layout-oriented fuzzing to find a diverging path that will free the exceptional object, and trigger an exploitable state. Given that the `glibc` library uses a double-linked list to maintain objects, unlinking a node from this list (due to certain memory operations) will update forward and backward nodes' pointers, causing an unintended memory write operation. This is known as `unlink attack` [6].

However, to successfully exploit such states, we have to arrange the heap layout, with heap Fengshui and other techniques, which is out of the scope of this paper. However, with the inputs generated by Revery, experts could manually massage the heap layouts and write an exploit much quicker.

6.2.3 **Failed Cases.** As aforementioned, Revery cannot guarantee to generate working exploits or trigger exploitable states. In our experiments, Revery failed for 5 programs.

Limitations of Vulnerability Detection. For some of the programs, Revery fails to detect the security violations. For example, `SimpleMemoPad` has a buffer overflow inside objects, i.e., it will corrupt the neighbor data fields rather than neighbor objects. Revery currently only

[1] We did not evaluate Revery on CGC programs which have heap-based vulnerabilities or real world programs, because the binary analysis engine angr [31]'s constraints solving ability is not enough for complex programs. And we are still working on it.

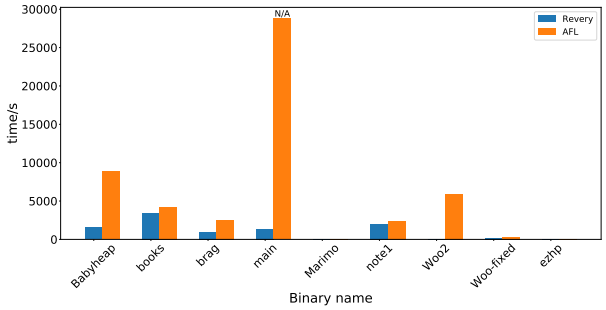


Figure 10: Time interval for finding first exceptional object by Revery, comparing to AFL.

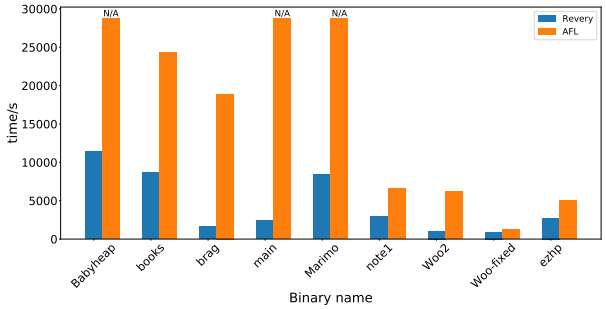


Figure 11: Time interval for finding exploitable states in diverging path by Revery, comparing to AFL.

supports object level corruption detection. We leave it as a future work to support detection of in-object buffer overflow.

Limitations of Angr. Our solution Revery relies on angr [31] to perform symbolic execution. Angr emulates all syscalls by itself, which has not fully implemented yet. Alternatively, angr rewrites library functions in Python, and hooks the original functions. However, this is far from finished too. As a result, angr cannot support most real world programs. This is also the major reason why we only evaluate Revery on CTF programs.

For example, to exploit childheap, some special characteristics of the fgets function are required. This function is hooked by angr but the required features are not properly implemented. So Revery is not able to find a way to exploit the vulnerability.

6.3 Efficiency of Layout-oriented Fuzzing

We further evaluated the efficiency of Revery in terms of diverging path exploration and exploitable states searching. We compared our layout-oriented fuzzing with the original fuzzer AFL.

Figure 10 shows the time interval used by Revery and AFL to find the first input that hits all instructions in layout-contributor slice. On average, Revery is 122% faster than AFL.

Revery also spends less time than AFL to find an exploitable state in diverging paths. As shown in Figure 11, AFL failed to find exploitable states for 3 programs in 8 hours. By contrast, Revery has found exploitable states for all the programs. For programs that both AFL and Revery succeed, Revery is 247% faster than AFL on average.

In short, with layout-oriented fuzzing, Revery could find diverging paths and exploitable states much faster than AFL.

Name	Vul Type	Revery Gen. Time (s)	Path Reuse Rate	Revery EXP. Work	SYMBEX. Gen. Time(s)	SYMBEX. EXP. Work
shop 2	UAF	238	100%	YES	Failed	NO
note2	BOF	70	100%	YES	Failed	NO
ezhp	BOF	56	98.0%	YES	Failed	NO
fb	BOF	60	85.1%	YES	Failed	NO
note3	BOF	83	84.1%	YES	Failed	NO
main	UAF	146	71.1%	YES	>4hours	Unknown
stkof	BOF	208	65.5%	YES	Failed	NO
marimo	BOF	264	62.2%	YES	Failed	NO
simplenote	BOF	263	41.9%	YES	Failed	NO
babyheap	UAF	442	27.8%	YES	Failed	NO
note1	BOF	161	84.0%	YES	412	YES
b00ks	BOF	81	83.3%	YES	91	YES
woO2	UAF	38	22.7%	YES	39	YES
woO2_fixed	UAF	38	22.7%	YES	38	YES

Table 2: Comparison with symbolic execution

6.4 Efficiency of Control-Flow Stitching

Given the candidate exploitable states, Revery utilizes a novel control-flow stitching solution to generate inputs to trigger both the vulnerability and exploitable states. In theory, symbolic execution could be used solely to explore paths from the vulnerability point to the exploitable states. To compare the efficiency between them, we thus evaluated Revery and a strawman symbolic exuection tool SYMBEX based on angr.

6.4.1 Overall Results. Table 2 shows the evaluation results on 14 programs which angr is able to handle. Revery could generate EXP inputs to trigger exploitable states for all 14 programs in minutes. But SYMBEX could only solve 4 of them. The exploitable points of these 4 programs are right after the vulnerability points and before the crashing points, and thus require no efforts to explore paths. SYMBEX failed to solve the program main in four hours, and failed for the rest 9 programs.

Path Reusing Rate. We use path reusing rate to assess the quality of stitching points that Revery found. This rate is computed based on the count of basic blocks reused from the diverging path, compared to the count of basic blocks in the exploitation path. A higher reusing rate indicates that the stitching point is better for exploit generation. As shown in the table, more than half of the programs has a path reusing rate higher than 60%.

Failure Analysis. SYMBEX failed for 9 programs. We pointed out that, traditional symbolic execution is unable to infer some exploitable state constraints and thus fails to generate exploits. As discussed in Section 5, Revery could infer these constraints from the diverging path. In the following, we take two programs babyheap and marimo as examples.

Listing 1: Code fragment of babyheap

```
1    i = get_int_from_user()
2    team = team_list[i];
3    if ( team ) manage_team(team);
```

```
1.  puts("Select number or [B]ack");
2.  printf(">>");
3.  scnaf("%c",&tmp)
4.  index = tmp-0x30;
5.  edit_marimo(marimo_list[index])
```

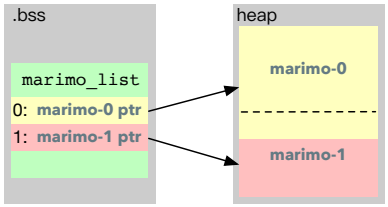


Figure 12: An CTF program `Marimo`. Data constraints is essential for generating a heap overflow exploit.

Example 1: `babyheap`. As shown in Listing 1, the target exploitable state is in the function `manage_team`, which could only be triggered if the the argument `team` is not NULL. But this value is retrieved using a symbolic index from the array `team_list`. Traditional symbolic executions, e.g., SYMBEX, will concretize the symbolic index, and in most cases will get a NULL pointer from the array. As a result, it could not trigger the exploitable state.

Example 2: `marimo`. Figure 12 illustrates the vulnerability of `marimo`. There is an overflow in the `marimo-0` object, which corrupts the object `marimo-1`. If and only if the object `marimo-1` is passed to `edit_marimo` function, the exploitable state could be triggered. However, angr does not know the knowledge and gets a random value from the solver.

On the other hand, `Revery` knows this knowledge from the crashing path, and adds the extra constraint of the exploitable state. So it could generate exploits successfully.

7 RELATED WORK

7.1 Automatic Exploit Generation

`Revery` aims at automatic exploit generation, which is still an open challenge. A few number of solutions have been proposed.

7.1.1 AEG Based on Symbolic Execution. APEG [12] is the first automated exploitation solution based on patch analysis. AEG [9] develops a novel preconditioned symbolic execution and path prioritization techniques to generate exploits at the source code level. Mayhem [13], which is built based on the hybrid symbolic execution and memory index modeling techniques, can automatically generate exploits at the binary level.

These solutions symbolically execute the whole program and are not scalable in path exploration. Unlike `Revery`, they are unaware of exploitable state constraints. Moreover, they concretize symbolic indexes, e.g., Mayhem untilizes a prioritized concretization, but it could still lead to non-exploitable states. In addition, they are not able to handle heap-based vulnerabilities.

7.1.2 AEG Based on Crash Analysis. Sean Heelan [18] makes use of dynamic taint analysis and program verification to generate control-flow-hijack exploits based on the crashing PoC input. Similarly, starting from the crashing point, CRAX [20] symbolically executes the program to find exploitable states and automatically generates working exploits at the binary level.

These solutions only search the crashing paths for exploitable states. As aforementioned, exploitable states do not always exist in crashing paths. So they will be hindered by the exploit derivability issue. By contrast, `Revery` explores exploitable states not only in crashing paths but also in diverging paths.

7.1.3 Data-Oriented AEG. FLOWSTITCH [19] automatically generates data-oriented exploits, able to reach information disclosure and privilege escalation, by stitching multiple data flows without breaking the control flow.

Although it also uses stitching, it is quite different from `Revery`. First, it targets data-flow stitching, while the control-flow is intact, making symbolic execution easier. Second, it only produces exploits of data-only attacks, instead of control-flow hijacking attacks.

7.2 Directed Fuzzing

`Revery` utilizes fuzzing to explore diverging paths. There are many advances in this field in recent years.

7.2.1 Coverage-Guide Fuzzing. There are many works which aim to increase code coverage of fuzz testing, called coverage-guide fuzzing. AFL [37], libFuzzer [27], honggfuzz [35], AFLFast [11], VUzzer [24] and CollAFL [16] are some state-of-the-art coverage-guided fuzzers. In general, they prioritize the seeds with higher code-coverage for further mutation. However, they do not target specific code or memory states, and thus are not efficient in exploring diverging paths which must satisfy some requirements.

7.2.2 Target-Directed Fuzzing. The most similar work to our focus is AFLGo [10], a greybox fuzzing tool. AFLGo[10] prioritizes seeds that are closer to predetermined target locations, enabling efficient directed fuzzing. But a diverging path consists of multple target points. And AFLGo is not effective in exploring diverging paths which have multiple target points. `Revery` guides a fuzzer with layout-contributor slice to explore diverging paths and search for exploitable states efficiently.

7.3 Vulnerability Detection

`Revery` utilizes memory tagging to detect security violations. There are many sanitizers [2, 25, 29, 33] proposed for this purpose. For example, ASAN [29] pads objects with redzones and places freed objects into quarantines, able to detect spatial and temporal violations. SoftBound [21] records base and bound information for every pointer as disjoint metadata to enforce completely spatial memory safety for C programs. CETS [22] uses a key and lock address with each pointer in a disjoint metadata space and checks pointer dereferences to enforce temporal safety for C programs.

All of these solutions will slightly change the memory layout of target programs as a result of the instrumentation and thus are not suitable for exploit generation. By contrast, `Revery` utilizes a shadow memory to track the tags of pointers and the status of heap objects non-intrusively.

8 DISCUSSION

AEG is an open challenge. `Revery` only moves one step towards this goal. It has many challenges, including but not limited to:

- **Advanced Defenses.** More and more defenses are proposed and deployed in practice, in order to stop popular attacks. These defenses not only raise the bar for human attackers, but also hinder automated solutions. For example, `Revery` could not bypass ASLR because it lacks the ability of information disclosure. It could trigger exploitable states for 5 of 19 programs, but not able to generate working defenses, because of the sanity checks deployed in heap allocators.
- **Heap Layout Massaging.** A large number of heap-based vulnerabilities could only be exploited in specific memory layouts. Due to the complexity of memory allocators and the program behavior, it is very challenging to generate inputs to build memory layouts as expected.
- **Combination of Multiple Vulnerabilities.** In practice, a successful exploit usually require multiple vulnerabilities. We have to assemble different vulnerabilities and utilize their corruption effects to craft a final exploit.
- **Program Comprehension and Analysis.** To successfully exploit a program, it is necessary to understand the program behavior, e.g., what input will cause what output, and make dynamic decisions at runtime. In addition, few program analysis solutions could extract such information. As aforementioned, the widely used symbolic execution has many limitations too.

9 CONCLUSION

Existing AEG solutions are facing the challenges from exploit derivability issue, symbolic execution bottleneck and heap-based vulnerabilities. We proposed a solution `Revery` able to search exploitable states in diverging paths rather than crashing path, with a novel layout-oriented fuzzing and a control-flow stitching solution. It could trigger both vulnerabilities and exploitable states for a big portion of vulnerable applications. It could also successfully generate working exploits for certain vulnerabilities. It has moved one step towards practical AEG. But there is a long way to go.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive comments. This work is supported in part by Beijing Municipal Science and Technology Project (No.Z181100002718002), National Natural Science Foundation of China (No. 61572481 and 61602470, 61772308, 61472209, 61502536, and U1736209), and Young Elite Scientists Sponsorship Program by CAST (No. 2016QNRC001).

REFERENCES

[1] 2018. CTF TIME. https://ctftime.org. (2018). Online: accessed 01-May-2018.
[2] 2018. DataFlowSanitizer. https://clang.llvm.org/docs/DataFlowSanitizerDesign.html. (2018). Online: accessed 01-May-2018.
[3] 2018. !exploitable Crash Analyzer. http://msecdbg.codeplex.com/. (2018). Online: accessed 01-May-2018.
[4] 2018. OSS-Fuzz: Five Months Later, and Rewarding Projects. https://testing.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html. (2018). Online: accessed 01-May-2018.
[5] 2018. Rex - Shellphish's automated exploitation engine . https://github.com/shellphish/rex. (2018). Online: accessed 01-May-2018.
[6] 2018. Unlink Exploit . https://heap-exploitation.dhavalkapil.com/attacks/unlink_exploit.html. (2018). Online: accessed 01-May-2018.
[7] S. Andersen and V. Abella. 2004. Data Execution Prevention: Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies. http://technet.microsoft.com/en-us/library/bb457155.aspx. (2004).
[8] Thanassis Avgerinos, Sang Kil Cha, Brent Lim, Tze Hao, and David Brumley. 2011. AEG: Automatic exploit generation. In *Network and Distributed System Security Symposium*.
[9] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. 2014. Automatic exploit generation. *Commun. ACM* 57, 2 (2014), 74–84.
[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2329–2344.
[11] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1032–1043.
[12] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *Proceedings of the 29th IEEE Symposium on Security & Privacy*. Oakland, CA.
[13] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 380–394.
[14] DA DARPA. 2014. Cyber grand challenge. *Retrieved June* 6 (2014), 2014.
[15] Thomas Dullien and Halvar Flake. 2011. Exploitation and state machines. *Proceedings of Infiltrate* (2011).
[16] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. CollAFL: Path Sensitive Fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, Vol. 00. 660–677. https://doi.org/10.1109/SP.2018.00040
[17] Liang He, Yan Cai, Hong Hu, Purui Su, Zhenkai Liang, Yi Yang, Huafeng Huang, Jia Yan, Xiangkun Jia, and Dengguo Feng. 2017. Automatically assessing crashes from heap overflows. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 274–279.
[18] Sean Heelan. 2009. *Automatic generation of control flow hijacking exploits for software vulnerabilities.* Ph.D. Dissertation. University of Oxford.
[19] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. 2015. Automatic Generation of Data-Oriented Exploits.. In *USENIX Security Symposium*. 177–192.
[20] Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang, Chung-Wei Lai, Han-Lin Lu, and Wai-Meng Leong. 2012. Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. In *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*. IEEE, 78–87.
[21] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Intl. Conf. on Programming Language Design and Implem.*
[22] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C.
[23] PaX-Team. 2003. PaX ASLR (Address Space Layout Randomization). http://pax.grsecurity.net/docs/aslr.txt. (2003).
[24] Sanjay Rawat, Vivek Jain, Ashish Kumar, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Network and Distributed System Security Symposium*.
[25] Alexey Samsonov and Kostya Serebryany. 2013. New features in AddressSanitizer. (2013).
[26] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2011. Q: Exploit Hardening Made Easy.. In *USENIX Security Symposium*. 25–41.
[27] Kosta Serebryany. 2016. Continuous Fuzzing with libFuzzer and AddressSanitizer. In *Cybersecurity Development (SecDev), IEEE*. IEEE, 157–157.
[28] Kostya Serebryany. 2017. OSS-Fuzz - Google's continuous fuzzing service for open source software. (2017).
[29] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *the 2012 USENIX Annual Technical Conference*. 309–318.
[30] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. 2018. Memory Tagging and how it improves C/C++ memory safety. *CoRR* abs/1802.09517 (2018). arXiv:1802.09517 http://arxiv.org/abs/1802.09517
[31] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 138–157.
[32] Alexander Sotirov. 2007. Heap feng shui in javascript. *Black Hat Europe* (2007).
[33] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*. IEEE, 46–55.
[34] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *NDSS*, Vol. 16. 1–16.
[35] Robert Swiecki. 2016. Honggfuzz. *Available online a t: http://code. google. com/p/honggfuzz* (2016).

[36] Julien Vanegue. 2013. The automated exploitation grand challenge. In *presented at H2HC Conference*.
[37] Michal Zalewski. 2018. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/. (2018). Online: accessed 01-May-2018.