Soft Comput (2017) 21:3401–3414
DOI 10.1007/s00500-015-2017-6

METHODOLOGIES AND APPLICATION

WhirlingFuzzwork: a taint-analysis-based API in-memory fuzzing framework

Baojiang Cui[1] · **Fuwei Wang**[2] · **Yongle Hao**[3] · **Xiaofeng Chen**[4]

Published online: 14 January 2016

Abstract Fuzz testing is widely used as an automatic solution for discovering vulnerabilities in binary programs that process files. Restricted by their high blindness and low code path coverage, fuzzing tests typically provide quite low efficiencies. In this paper, a novel API in-memory fuzz testing technique for eliminating the blindness of existing techniques is discussed. This technique employs dynamic taint analyses to locate the routines and instructions that belong to the target binary executables, and it consists of parsing and processing the input data. Within the testing phase, binary instrumentation is used to construct loops around such routines, in which the contained taint memory values are mutated in each loop. According to experiments using the prototype tool, this technique could effectively detect defects such as stack overflows. Compared with traditional fuzzing tools, this API in-memory fuzzing eliminated the bottleneck of interrupting execution paths and gained a greater than 95 % enhancement in execution speed.

Communicated by V. Loia.

✉ Baojiang Cui
cuibj@bupt.edu.cn

Fuwei Wang
fuwei.wfw@alibaba-inc.com

Xiaofeng Chen
xfchen@xidian.edu.cn

1 Beijing University of Posts and Telecommunications and National Engineering Laboratory for Mobile Network Security, Beijing, China

2 Security Department, Alibaba Group, Hangzhou, China

3 China Information Technology Security Evaluation Center, Beijing, China

4 School of Telecommunications Engineering, Xidian University, Xi'an, China

Keywords Software testing · Fuzz testing · Taint analysis · Control-flow hijacking

1 Introduction

In recent years, data-processing programs have been widely used in the field of office automation. In addition to the increasing demands of software and the shortened development cycles, commodity applications are facing increasingly greater risks of exploitable vulnerabilities. Among these vulnerabilities, the most common type is file format vulnerabilities. In general, such vulnerabilities are caused by the lack of a corresponding strategy for handling irregularly formatted input files; thus, exceptions may be triggered within special execution paths.

There are two models for discovering software vulnerabilities: white-box and black-box. The mainstream white-box tests are source code oriented, on which syntax analysis and defect pattern matching are conducted. In contrast, black-box tests are designed for third-party analysts to perform peripheral tests on binary executables. The primary method for mining file format vulnerabilities is to externally construct malformed sample files to pass into the program with the expectation that more combinatorial paths, which may contain logically defective branches, be touched during the processing procedure. A representative implementation technique is fuzz testing, which generates new files or mutates normal seed files to enforce brute-force path traversals. The key of this technique is its intervention-free randomness, which enables many unreasonable exceptions to be triggered that are difficult to construct through manual analysis. However, the randomness of this technique is accompanied by blindness, in which violation of the format stipulations will primarily lead to failures of the initial file

validity check and in which program execution will terminate during early stages. As an improvement, the file format fuzzing technique introduces corresponding format specifications and mutates seed files while not violating the normal structure, which effectively reduces the blindness. However, the fine-grained specifications are difficult to acquire, and constructing valid samples can be a quite complicated and empirical process. Moreover, to increase the testing efficiency against the low detection ratio of fuzz testings, large-scale distributed testing frameworks are widely used in industries, which require an unlimited amount of computing resources.

To obtain more accurate profiles of the behaviors of programs, a wide range of studies have emerged in the academic literature, and these studies are divided into data-flow and control-flow analyses. Having extracted the processing chain of specific data throughout the process, data-flow analysis further infers the functions of each binary module, and the most representative technique is dynamic taint analysis (DTA) (Newsome and Song 2005). The elementary form of control-flow analysis is to statically reverse engineer the target executable, abstract all possible execution paths and match them to typical patterns of vulnerabilities; furthermore, the symbolic execution technique (Schwartz et al. 2010) takes the key branches along the control flow as constraints, which are solved to obtain the set of input data that lead to a specific path. However, the limitations of both analyses are obvious: taint analysis is designed as a supporting technique but is not directly oriented to mining vulnerabilities, whereas the application of symbolic execution is confronted with some problems, such as path explosion.

In this article, we present a novel fuzz testing technique. The key concept of this technique is, in contrast to conventional methods, accurately targeting the in-memory data that originated from external inputs and mutating them to affect a limited spectrum of routines. As a preliminary step, we first utilize the results of a taint analysis and obtain a complete list of functions and corresponding instructions that involve the processing of taint data. During the testing phase, the execution path is elaborately controlled: instrumentations on the entry and exits of each *taint* function are set to artificially create loops around them, and within each loop, some specific *taint* memory operands are mutated, which are specified by the list of taint instructions. In this manner, each function that is relevant to input parsing is sufficiently tested with all possible conjunctions of the input data that it touches. Compared to traditional testing methods, there are at least two advantages of this in-memory fuzz testing technique:

- While the basic *fuzzy* property is maintained, the testing procedure bypasses possible data validation steps and targets potentially vulnerable functions directly at a deep level of a control flow or a call stack, which leads to a considerably greater code coverage than traditional fuzz testings. Moreover, because the target functions are executed and tested in the actual execution contexts and environments, those unrealistic testing inputs are filtered out, in contrast with static analysis, and the complicated solving of direct external data from input to exceptions is no longer required.
- Because every testing round is actually a loop of a specific function, the entire testing course does not need to restart the target process, and any procedures during the process that are not relevant to taint parsing are not repeated during subsequent executions, such as the program initialization and receiving external data from I/O or sockets. In this way, the testing efficiency is greatly enhanced, by at least 95 % according to the following evaluations.

This article is divided into five parts. Section 2 discusses the demands and design of a base taint analysis framework. Section 3 presents the technical principles and proposes two designs of in-memory fuzzing based on virtualization and debugging. Section 4 presents the details of our in-memory fuzz testing framework *WhirlingFuzzwork*. Section 5 presents the evaluation methods and results to demonstrate the effectiveness and efficiency of the proposed testing framework. Finally, in Sect. 6, we present the related works and draw a brief conclusion to the entire work.

2 Exploration of taint API in-memory fuzzing

2.1 Principle

When referring to the behaviors of a computer program, three elements are included: memory, registers and external storage. Most runtime data of a running program are stored in memory owned/shared by the corresponding process; in finer granularity, the data passed in and out of an instruction of the program are stored and transferred via its memory and register operand. Such memory and register data make up the base of *runtime state* of a running program. Besides, data could be loaded from disk files or network streams into memory as input and written back as output. In a word, if two processes owns exactly the same in-memory instructions and data, while the processor registers and external resources are also made the same, they are technically *identical*. In other words, if we save all these states of a process and then restore them on a different spot, we could make it run identically thereafter. That is the main concept of deterministic replay of programs, which has been intensively studied in

recent years, and some designs and prototypes seem quite promising for applications (Srinivasan et al. 2004; Patil et al. 2010).

The basic idea of in-memory fuzzing is motivated by the technique of deterministic replay of programs. Since the state of a process can be saved and restored at any runtime point (namely*checkpoint*), it is made possible to repeatedly execute some specific program procedure, starting from the same process context, with its behaviors guaranteed to be identical if given the same input data from the entry point. If the in-memory input data are mutated on purpose, all possible execution paths through that procedure can be traversed and tested for potential defects, which makes up the fuzz testing course. Such mutable in-memory data are called taint data, which originated from external source hence artificially controllable.

The concept of in-memory fuzzing first appeared in Michael Sutton's monographs (Sutton et al. 2007) related to fuzzing tests. In-memory fuzzing is intended to artificially create loops to call processing functions and directly test the robustness of data-processing modules whose input data are difficult to repeat, such as network data flow. Based on this concept, the Corelan Team developed a prototype system (Team 2010) that loop executes each function and mutates the function parameter list in each loop.

Figure 1 illustrates the principle of in-memory fuzzing. We divide the data-processing program, whose execution process only involves the data flow, into two parts: the data-loading and the data-processing phases. The different input data are always loaded by the same loading procedure, are checked at the initial step of the processing phase, and later undergo a series of refining processes. In traditional fuzzing tests, the fuzzing tool mutates the input data (normally in forms of files and network streams) and restarts the entire process in each test round. It repeats the data-processing-independent process, including the data-loading process, and often interrupts the execution path because the mutated input data cannot proceed through the data check steps or because exceptions occur in the refining process function. Additionally, the API in-memory fuzzing test normally executes the program after the data-loading process and data-checking process. Then, the fuzzing test loop executes each processing function a limited number of times by setting the function context return to the entry at the end of the loop and mutates the copy of input data in the memory to traverse different path combinations inside the function.

The previous two studies demonstrate the advantages of the in-memory fuzzing test, which can focus on the test target by bypassing the data-processing-independent process and enumerates paths in a specific function range and target process. The in-memory fuzzing test ensures that the data mutation step occurs after the execution path reaches a certain depth. However, both designs still possess theoretical defects. The former study requires an extensive early manual analysis to accurately strip out the data-processing module and lacks the specification of mutated memory. It is also not feasible to artificially add the loop structure into the binary program. Therefore, the former study is merely an exploration of the concept. The latter study only mutates parameters in function memory, which are uncertainly controlled by external input data. In other words, most of the input data cannot be mutated.

This study solves the aforementioned problems using binary instrumentation and the taint analysis technology. Our fuzzing tool can accurately locate functions that process the taint data by utilizing the result of a taint analysis, and it mutates the specified tainted memory operand in these functions. Therefore, the grain size narrows the taint data on the instruction operand.

In order to undertake an API in-memory fuzzing, we need to first locate the *tainted* memory segments which could be mutated during runtime, and then to monitor and hijack the control flow; more precisely, we should be able to memorize the status on some specific process points and restore the context to it at any time. In this section we first introduce the technique of dynamic taint analysis (DTA) and its adopted implementation. Then we will discuss the possible architectures of different designs of API in-memory fuzzing, utilizing system-wide virtualization and process debugging API, respectively.

2.2 Preliminary taint analysis technique

Dynamic taint analysis (DTA) is a featured technique that monitors the execution of a target binary program; detects the data input from some certain interfaces, which are denoted as *taint* sources; and walks through every instruction executed to track the accessing, propagation and elimination of taint data. The result of a DTA is typically a taint trace that shows the processing chain on each input byte. As a preparatory base for illustrating the targets of the fuzz testing, a practical taint analysis framework is implemented.

2.2.1 Off-line DTA framework FlowWalker

DTA is normally conducted on assembly instruction granularity to implement fine-grained data tracking. Among the full set of instructions, integral, float point and bitwise arithmetical operations commonly result in partial overlapping and elimination of data from different sources. For instance, when the two operands hold the base and relative offset data of a payload in a file, an `ADD` instruction is used to sum them to obtain the absolute offset. In this case, both taint source tags should be present in the record of the taint sta-

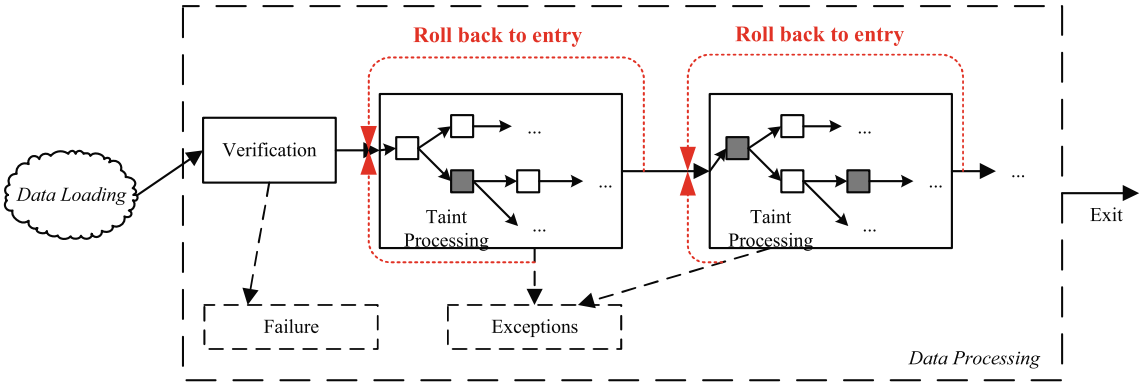

Fig. 1 Structure of taint manifest

tus of the destination operand; otherwise, an interception of one branch of the taint trace will be introduced. Unlike other DTA tools that have to adopt single-tag taint regulations for low execution overhead, we utilize multi-tag taint algebra to represent the taint properties in the stand-alone taint-tracking module.

The existing DTA tools are divided into virtualization-based and instrumentation-based types. For both types, the tool needs to monitor the opcode type and the operand status of each instruction along with the dynamic execution; meanwhile, a global record of taint storage is frequently queried and updated, which imposes infeasible time and space overheads on the target process. In our preliminary research, we borrowed ideas from the TaintScope (Wang et al. 2010) project and developed an off-line DTA framework, FlowWalker (Cui et al. 2013), which integrates the dynamic execution with a stand-alone data flow-tracking module. In practice, FlowWalker has been demonstrated to meet the requirements for analyzing extensive software.

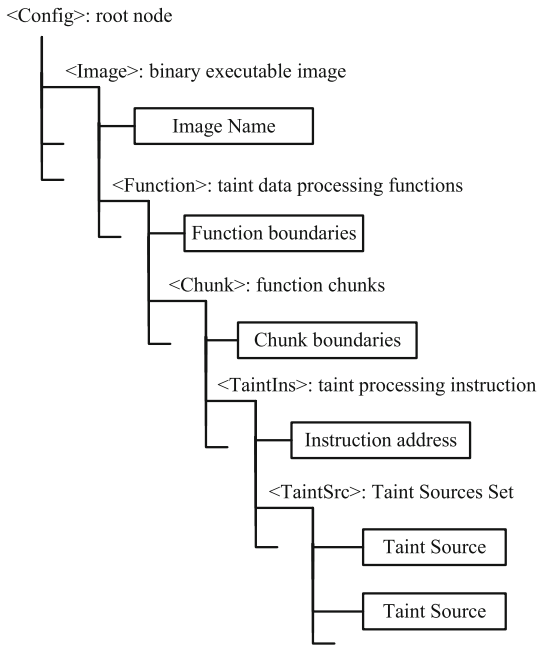

Fig. 2 Structure of taint manifest

2.2.2 Interface between DTA and fuzzing

Taint analysis orients all taint bytes and outputs a complete data flow trace for each. Such a *taint trace* is a sequence of operational taint nodes, which consist of a 3-tuple with a time-stamp, the binary image it belongs to and its instruction address. To conform the taint information to function boundaries, IDA scripts are utilized to load and incorporate taint traces into lists of taint functions in the form of an XML manifest, as illustrated in Fig. 2.

2.3 Fuzzing with virtualization

One most important feature of virtual machines is the snapshot technique. By taking and restoring system-wide snapshots, the whole environment of the system as well as the target process could be restored to that of some other time point, or namely *checkpoint*. This is made possible when using an extendible and programmable virtualization platform, such as the QEMU (Bellard 2005) and Xen (Barham et al. 2003). Besides, deterministic replay based on virtualization has drawn quite much attention (Dunlap et al. 2008; Laadan et al. 2010), which could provide concrete back-ends for in-memory fuzz testing purpose. In this part we are going to discuss some abstract requirements of implementing an in-memory fuzzing tool based on a virtual machine; as for some technical descriptions, we will borrow the concepts of QEMU as a representative.

A virtual machine is divided into two layers: a guest-OS is the actual virtualized operating system running on an emulated CPU, and a VM-monitor in charge of monitoring the system, which also communicates with customized plug-ins to undertake user specified instructions. In this case, we could

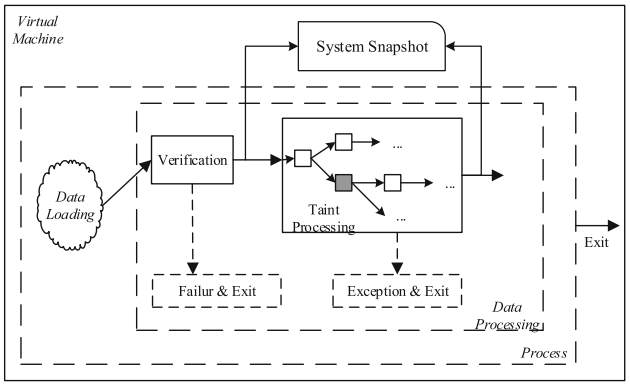

Fig. 3 Structure of virtualization-based fuzzing

integrate the fuzzing policy into a plug-in to take effects via controlling the monitor. The overall architecture is illustrated in Fig. 3.

2.3.1 Artificial cycle control

In order to enable cyclic execution around each taint function, the VM-monitor needs to locate them during runtime first. This could be done by patching the instruction dynamic translation, so that the taint function entry instructions could be instantly located by their addresses.

In order to enforce manual control on the execution path, the fuzzing tool could employ the inspection-spot snapshot. When the entry instruction of some taint function is executed, the system as well as the process is halted to take a snapshot of the memory and CPU registers. Afterwards, when one return instruction of that function is encountered, this snapshot is restored to ensure the consistency of runtime environment for rounds of tests of the same function.

2.3.2 Internal and external control

A process involves both user-space and kernel-space objects and events. To ensure both environments are kept unchanged for fuzzing, the fuzzer needs extra internal and external control over them.

The internal control deals with involved kernel objects, such as the status of locks and mutex. By restoring inspection-spot along with memory snapshots, such objects are automatically restored without any other disposition, which is an obvious advantage. The external control deals with all resources outside the process, such as the files/registry entries being written within the scope of the tested function. Such changes need to be recorded while the first run of the target taint function, and if there are any, a complete disk snapshot is additionally demanded to roll them back.

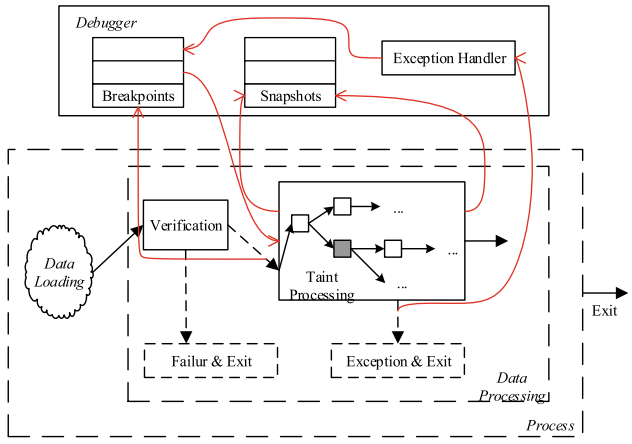

Fig. 4 Structure of debugging-based fuzzing

2.3.3 Mutation

The mutation of in-memory taint data is done for each instructions which accessed them. In this sense, the mutating operation must be carried out on the instruction translation mechanism of the VM. If the VM supports dynamically modifying the memory data in the translation step, the tool only needs to monitor the execution of specified instruction addresses and correspondingly mutate the data of their memory operand; otherwise, the modification could only be conducted by locating the position of the tainted memory in the dumped memory snapshot and changing it to be flushed back to the context of the next run.

2.4 Fuzzing with debugging

As most of the other fuzzing techniques and tools chose, the most straight-forward way to fuzz a program is to monitor it via a debugger, utilizing system debugging API to monitor the status of processes. With the help of some debuggers which provide the ability to modify running contexts and rich programming support, such as PyDbg (Amini 2006), it is also achievable to implement in-memory fuzzing. A design of such a fuzzing architecture is illustrated in Fig. 4.

2.4.1 Artificial cycle control

The cycle control could be achieved by setting up debugging breakpoints at the entry and exit of the target program modules after they are loaded into memory. Once an entry-point breakpoint is triggered on, the process is halted and the debugger takes control to manipulate the information of the function under surveillance. Correspondingly, the breakpoints at each return instruction of that function is triggered to enable wind-up tasks, including rolling the process back to the function entry and restoring snapshots.

To back up the process status at the entries of functions, the fuzzer needs to dump the raw data of the whole memory space, which is an existing functionality of some kernel-level debuggers like WinDbg, originally intended for system crash analysis.

2.4.2 Internal and external control

The control over internal and external object remains as the biggest problem of a debugging-based implementation. Since a debugger normally operates on Ring-3 level programs, any changes to the kernel are invisible and untraceable. For example, when the program opens a file and a function reads some data from a certain position in the file object, the file pointer moves that amount of bytes forward correspondingly. If the file pointer is not accordingly processed, the same function will read out different data every time it is rolled back and gets executed.

2.4.3 Mutation

Having obtained the result of taint analysis, we could tell the whole set of instructions which touched the tainted memory along the normal procedure of processing the normal taint data. However, as we changed the execution path, not all of these instructions are touched, and it would cause a fault if we modify all the data of these memory operands at the entry of one testing round. Under the debugger, the program could be inspected step by step. Only if a taint instruction is to be executed, its memory operand gets modified right before that.

2.5 Practice considerations

As above, we discussed the general requirements of an API in-memory fuzzing functionality and analyzed the key points of feasible implementations of the prototypes under virtualization and debugging. However, both techniques face some solid obstacles in practice:

- While a virtual machine could serve most demands, the current products under development do not provide easy-to-use programming interfaces, and some functions need to make in-depth changes to the source codes of the underlying VM. Besides, since the checkpointing of a virtual machine is system-wide, the snapshots generation and restoration consume an infeasible amount of time or even need restarting the VM, which makes it a heavy-weight back-end and break the structure of process-oriented in-memory fuzzing.
- Compared to virtual machines, debuggers lack some key functionalities such as the ability to restore any changes of internal objects and external resources. This is a major drawback of debugging API that the control over the debugged program is limited, and the support of programming complicated control instructions is hardly realizable. Moreover, the program behaviors may be different under debugging if there is an anti-debugger setting.

Because of these limitations, it is quite hard to implement a fully functional and practical prototype of in-memory fuzzing tool based on virtualization or debugging alone. For the same reason, we turned to binary instrumentation technique and implement WhirlingFuzzwor, which we are going to discuss in the following section.

3 WhirlingFuzzwork: design and implementation

In order to get rid of the problems in implementing with either virtualization or debugging techniques, we chose a third solution to implement our fuzzing tool: dynamic instrumentation. A binary instrumentation platform owns the advantages of both former techniques: it deals with the original program instruction by instruction in a just-in-time (JIT) compiler way, so we could inspect the process at quite fine granularity; The target program is executed in a separate VM kernel and has all its system calls, events and threads dispatched, which could be taken under control of user-defined plug-ins; the memory management is taken over by the VM, so it is feasible to modify the data of tainted memory.

Utilizing the Pin (Luk et al. 2005) binary instrumentation platform, the fuzzing tool can hijack and take over the control flow of the target program. Moreover, it has been intensively studied by Intel group to achieve deterministic replay of program execution on top of Pin (Patil et al. 2010), which gives a concrete proof of the availability. However, the PinPlay extension serves general replay and analysis purpose with some performance overhead for exchange, and its programmable interfaces do not match all the needs of in-memory fuzzing control, so we designed and developed the prototype based on Pin directly. We implement several function modules, including appointed instrumentation, program breakpoint and real-time snapshot restoring, that can proceed through the whole loop control of execution paths.

Figure 5 depicts the design of the taint-analysis-based API in-memory fuzzing framework. The design of each module will be presented in the following.

3.1 Artificial cycle control

3.1.1 Taint call stack

The test target of in-memory fuzzing is the taint function. During execution of the target function, the framework creates loops to test the target function, which is called the *loop test body*, at specific times. However, the target function may

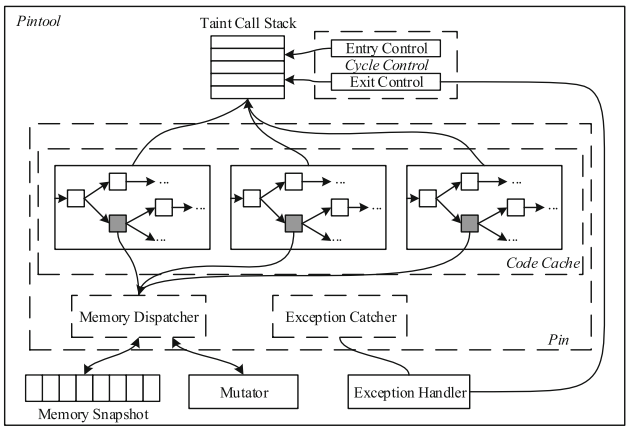

Fig. 5 Structure of whirlingFuzzwork

contain calls, nests or iterations, and thus, the taint data would propagate from the outer function to the inner function. If the loop test merely contains the outer taint function, the called inner functions would not be tested.

Therefore, in this study, the main body of the loop execution is the taint call stack. The taint function stack represents the call stack of the outermost taint function, and it provides the function layer identification for each instruction that operates on the taint data.

The taint call stack, which only considers the members of the taint function set, is similar in form to a real call stack. For instance, if the existing taint functions are A and B and the call stack is A→B→C, function C is located at layer 2 of the taint call stack.

Utilizing the taint call stack can help to effectively identify the functions in the hierarchy. For example, consider the generic call relation in Fig. 6.

There are five taint functions on the top three layers of the call stack in this call relation. Function F1 calls function F2 twice and iteratively calls function F5 once. We design a two-tuple $\langle Dep, Cnt \rangle$ as the taint function identification. Here, Dep represents the layer of the function being invoked in the call chain, and Cnt represents the number of times that the taint call stack reaches this layer. The identifications of function F2 being called both times are $\langle 2, 1 \rangle$ and $\langle 2, 2 \rangle$, and the identifications of function F5 that is iteratively called are $\langle 2, 3 \rangle$ and $\langle 3, 3 \rangle$. In this way, we can distinguish the execution order of the functions that called each other without storing the complete call stack.

The loop test body should execute the outermost taint function completely in each loop and mutate the touched taint data in the scope of the sub-function. The principle of the mutation order is from inner sub-function to outer parent function, from callee function to caller function. Assuming that all functions in Fig. 4 do touch the taint data and that the testing round count for each function is set to 2, the total number of loops will be 14. The data mutations are performed

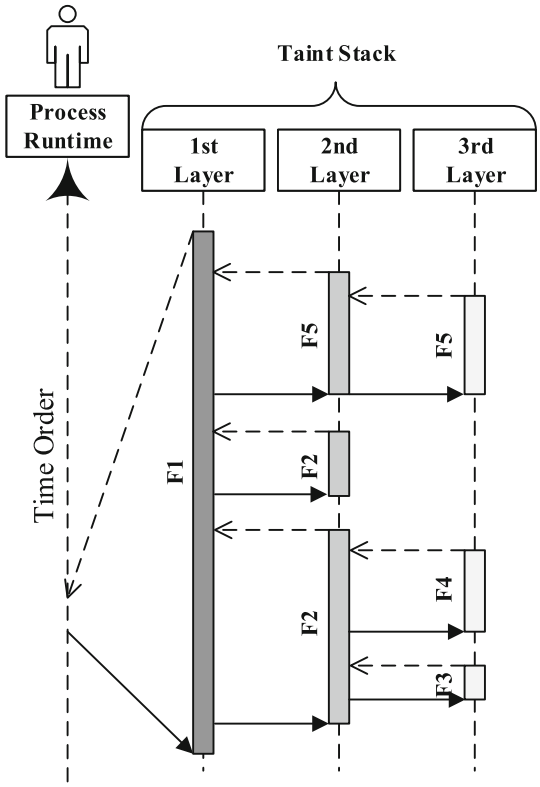

Fig. 6 Taint call stack example

aiming at call stack layer 3 (F5, F4, F3), the second layer (F5, F2, F2) and the first layer (F1).

3.1.2 Function entry control

The function entry control module is designed to maintain the taint call stack through instrumenting the entry instruction of each taint function. This module will push the sub-function, which gets called by its parent function, and taint function address on the stack.

The primary method to guarantee that the runtime environment is always the same at the start of every loop is to store and recover the state of registers. First, backup the context at the entry of the outermost taint function of the taint call stack. Afterwards, the control module can recover the context, which includes EIP and the frame pointer, at the end of a loop testing round and restart the function.

3.1.3 Function exit control

Built on binary instrumentation, the function entry and exit control modules comprise the looping test control structure of the call stack. The function exit control module contains the following three tasks.

Maintenance of taint call stack

After instrumentation and analysis of a RET instruction while the call stack is not empty, if the address of the RET instruction is located in the range of the taint function on the top of the stack, it would be popped. Furthermore, note that the function may return without any RET instruction because of the compiler's optimization. For example, the function whose return value is the other function's return value is often ended with a JMP instruction and jumps to the other function. In this case, it is necessary to judge the target address of JMP and pop the stack top if it is located outside of the scope of the current taint function.

Maintenance of loop structure

The pretreatment round is designed to determine whether the executed function processes the taint data. After the pretreatment round, the number of invoked taint functions within the taint call stack that touched taint data is counted, and the total number of testing rounds is then calculated given a certain testing loop number for each involved function. Once the taint call stack is empty, a test round would be finished, and then, the next round would be started after the selection of a function and instructions.

Recovery of runtime environment

Recovering the context of the entry at the end of one test round is the final step of the function exit control. Additionally, we design an after-treatment round that restarts the function without any mutation of the original taint data to continue execution after the testing of the function without mutated data.

3.2 Internal and external control

3.2.1 Memory snapshot management

Restoring the memory environment is also necessary for recovery of the runtime environment. Considering a 32-bit program, the snapshot of the complete user dynamic memory will require 2 GB of space. For a 64-bit program, it is not even possible to obtain a snapshot of the complete user dynamic memory.

Rather than a complete snapshot, in this study, we design a snapshot pool of the changed memory page to store the memory page that was written. After instrumenting of the write-memory instructions and obtaining the address of the written memory prior to instruction execution, the process copies and stores the memory page and adds it to the snapshot pool if the written memory is located outside of the memory pages in the snapshot pool. The data in the memory page snapshots are copied back to the original memory page after a loop test round.

In general, the memory range that is visited by one function is limited. The stack memory is limited in the range of the current stack frame, and the heap memory belongs to a finite number of heap objects. The ranges of the stack frame and heap objects are comparable. It requires a considerable amount of fragmented memory backup if the byte-grained memory snapshot is used. In contrast, using the page-grained memory snapshot significantly reduces the operating frequency that directly reads and stores memory data.

Because the memory snapshot module adds the instrumentations on all the instructions, which include the system call, the memory objects in the user mode will be recovered while restoring the memory environment, as will the heap table for the operation of heap memory allocation, such as the *malloc* function in the C runtime library that maintains it in memory. Therefore, recovery of the memory page snapshots also frees the memory that was allocated in this loop test round without extra garbage clean-up.

3.2.2 File object management

In contrast to memory snapshot management, the details of file objects are to be considered separately. The operating system provides two layers of file I/O mechanisms, including the runtime library and the system call. The I/O cache and the file pointer, which are maintained in the runtime library, will be returned to their original state after recovery of the snapshots. The cached data will be dropped, and the system call will be re-called. However, the file kernel object will be not in conformity with the file pointer, resulting in memory-read failure.

For this reason, file objects require particular recording and adjusting. In the pretreatment round, with the help of the *SetFilePointer* function, the file pointer offsets of the file kernel object should be recorded, and the position of the file-read pointer should be restored prior to beginning the next loop test round.

3.2.3 Monitoring record of process and exception

For the purpose of recording the test coverage in the whole test process, any loop test body that finished the test should be recorded by the monitoring record of process. The content of the record includes the source file position of the mutated memory byte and the memory value after mutation in each loop test round. The monitoring record of process will be used to generate deformity files based on the seed file.

In addition, the exception will be cached by the exception module of Pin. If an exception occurs, the instruction and context prior to the exception will be recorded. Additionally, the subsequent exception handling process will be skipped and the context and memory snapshots will be recovered for the next loop test round.

3.3 Mutation

3.3.1 Taint information import module

The taint information import module works prior to execution of the target program and binary instrumentation. This module reads the taint information from an XML list. Pin determines the base address when loading the module that contains taint functions according to the binary image that the module belongs to and then adjusts all the taint functions and taint instruction offsets to match the actual virtual address in the memory. The binary instrumentation will be directly set depending on the address.

3.3.2 Monitoring and mutation of taint instruction

The taint analysis can provide the list of functions that are involved with the taint data, but it cannot provide information on whether the taint function actually operates on the taint data during one specific processing. Hence, within the taint function, we add instrumentations on instructions that process the taint data and check whether the memory operand matches the actual taint data.

In the first execution round of the taint function, only the monitor for the taint instructions is mandatory. This monitor provides indices or arithmetic combination algebra of taint sources in the specific context in accordance with the imported taint information. If the memory data match the form of the taint data, the instruction will be added to the taint instruction set.

The process of cyclic mutation rounds of a taint function stack will directly mutate the memory operand if a current taint instruction is present in the taint instruction set that was generated in the first round.

Thus far, the prototype uses a combination of three mutation strategies, namely, randomization, boundary values and alternative taint source mutation. The alternative taint source mutation strategy replaces the original taint source with another taint source. This strategy can partially ensure the legitimacy of the mutated data. In the case of mutating data in PNG files or other multimedia files, the random mutation of data in the tag section often causes a type identification error. In contrast, the tag section can be mutated to a normal tag of an alternative segment following the taint source mutation strategy.

3.3.3 Generation and testing of mutated samples

The generation and testing of mutated samples are asynchronous with dynamic testing work to prevent a reduction in efficiency. The bytes in the source file positions that were recorded by the monitoring record module will be mutated to generate deformed sample files that may cause the exception. Afterwards, Python is utilized to call the target program from the input of sample files and monitor the recurrence of the exception.

4 Evaluation

To evaluate the theory of taint-based in-memory fuzzing and the implemented prototype, *WhirlingFuzzwork*, in this article, we present the evaluation methods and results to illustrate their correctness, effectiveness and efficiency. Because there are few similar fuzzing framework published so far, it is infeasible to reference some other works for comparison. Accordingly, we evaluate its correctness with sample programs, its effectiveness by code coverage, and efficiency by a comparison with traditional fuzz testing tools.

4.1 Correctness

4.1.1 Demonstration of program control

In the first part we experiment on the correctness of the cycle, internal and external control of our prototype WhirlingFuzzwork. First we build a target program with the following source code:

Listing 1 Code for testing internal control

```
#include <stdio.h>
#include <stdlib.h>
void test(int round, FILE* fptr) {
char cc=fgetc(fptr);
void* mc=malloc(0x100000);
cc+=0x20;
printf("
if (round<25)
test(++round,fptr);
}
int main() {
FILE* fptr=fopen("Y:\\test2.txt","r");
test(0,fptr);
return 0;
}
```

In this sample code, a file pointer object is passed into the taint function `test` as a parameter, and `test` is iteratively invoked 26 times to read an alphabet from the taint source file. Within each iteration, `1 MB` heap memory is allocated without being freed. During the experiment with WhirlingFuzzwork, we set the round number for each taint-function invocation to 2. After the testing, a log is generated recording 54 testing rounds; that is exactly the calculated number for the complete taint call stack scale plus an original and an ending round, which indicates that the cycle control is working as designed.

Because there is no explicit moving operation on the file pointer, the whole content of the source file is read into a cache all at once by default; after the first run of the 26 iterations of `test`, the file pointer of `fptr` had been moved to the end of the file, and any more reading operation would

cause an exception. During the experiment, the alphabet is output 54 times, which showed that the file object is properly restored so that in each testing round the function always reads from the same position of the file.

At the end of the original run of the target program, the process consumed about 27028 KB memory, which was approximate to the 26 MB memory allocated by the 26 invocations of test. During dynamic execution of the taint analysis, the process run under Pin consumed 40312 KB memory, which consisted of the overheads of both original process and instrumentation. During the experiment with WhirlingFuzzwork, the final memory consumption is 63526 KB. It demonstrates that the dynamically allocated heap memory blocks were properly freed after the corresponding testing round, otherwise the consumption should have grown over 54 × 26 MB = 1404 MB.

4.1.2 Demonstration against vulnerability

To demonstrate the correctness of the in-memory fuzzing theory, we conduct an experiment that targets a binary program with a stack overflow vulnerability, and its source code is as follows:

Listing 2 Code with stack overflow

```
#include <stdio.h>
const char header[]={"This is a fixed header. We \
are trying to make it a little longer so that a \
randomized fuzzer would quite likely to target \
its mutation in here hence cause a verification \
failure."};
int vul(char* payload) {
char buffer[8];
int index=0;
do {
buffer[index]=payload[index];
} while (payload[index++]!=0);
return index;
}
int wrapper(char* payload) {
int idx=0;
for (;idx!=179;++idx)
if (payload[idx]!=header[idx])
return 0;
return vul(payload+179);
}
int main() {
FILE* fptr=fopen("Y:\\test.txt","rb");
char src[1024]="\0";
int temp=0;
fread(src,1024,1,fptr);
temp=wrapper(src);
if (temp!=0)
wrapper(src+179+temp);
return 0;
}
```

This program aims to handle a 2-part text, in which each part consists of a specifier and an unfixed-length string that ends with a '\0' character. The raw content of the text is constructed as follows:

Listing 3 Taint source

```
0000h: 54 68 69 73 20 69 73 20 61 20 66 69 78 65 64 20 ; This is a fixed
0010h: 68 65 61 64 65 72 2E 20 57 65 20 61 72 65 20 74 ; header. We are t
0020h: 72 79 69 6E 67 20 74 6F 20 6D 61 6B 65 20 69 74 ; rying to make it
0030h: 20 61 20 6C 69 74 74 6C 65 20 6C 6F 6E 67 65 72 ;  a little longer
0040h: 20 73 6F 20 74 68 61 74 20 61 20 72 61 6E 64 6F ;  so that a rando
0050h: 6D 69 7A 65 64 20 66 75 7A 7A 65 72 20 77 6F 75 ; mized fuzzer wou
0060h: 6C 64 20 71 75 69 74 65 20 6C 69 6B 65 6C 79 20 ; ld quite likely
0070h: 74 6F 20 74 61 72 67 65 74 20 69 74 73 20 6D 75 ; to target its mu
0080h: 74 61 74 69 6F 6E 20 69 6E 20 68 65 72 65 20 68 ; tation in here h
0090h: 65 6E 63 65 20 63 61 75 73 65 20 61 20 76 65 72 ; ence cause a ver
00a0h: 69 66 69 63 61 74 69 6F 6E 20 66 61 69 6C 75 72 ; ification failur
00b0h: 65 2E 00 01 02 03 04 05 06 07 00 54 68 69 73 20 ; e.........This
00c0h: 69 73 20 61 20 66 69 78 65 64 20 68 65 61 64 65 ; is a fixed heade
00d0h: 72 2E 20 57 65 20 61 72 65 20 74 72 79 69 6E 67 ; r. We are trying
00e0h: 20 74 6F 20 6D 61 6B 65 20 69 74 20 61 20 6C 69 ;  to make it a li
00f0h: 74 74 6C 65 20 6C 6F 6E 67 65 72 20 73 6F 20 74 ; ttle longer so t
0100h: 68 61 74 20 61 20 72 61 6E 64 6F 6D 69 7A 65 64 ; hat a randomized
0110h: 20 66 75 7A 7A 65 72 20 77 6F 75 6C 64 20 71 75 ;  fuzzer would qu
0120h: 69 74 65 20 6C 69 6B 65 6C 79 20 74 6F 20 74 61 ; ite likely to ta
0130h: 72 67 65 74 20 69 74 73 20 6D 75 74 61 74 69 6F ; rget its mutatio
0140h: 6E 20 69 6E 20 68 65 72 65 20 68 65 6E 63 65 20 ; n in here hence
0150h: 63 61 75 73 65 20 61 20 76 65 72 69 66 69 63 61 ; cause a verifica
0160h: 74 69 6F 6E 20 66 61 69 6C 75 72 65 2E 00 0A 0B ; tion failure....
0170h: 0C 00                                           ; ..
```

When processing the text, any modifications on the header specifier would result in failure during the verification stage. If the payload length exceeds 8 bytes, the stack overflow vulnerability in function vul will be exploited, and the saved EBP register value and the return address on the top of the stack will be changed.

The original execution path is illustrated in the left of Fig. 7 in case that the number of testing rounds for each invoked tainted function is set to 10. There is a verification step deciding if the vul function is invoked. The path under testing is illustrated in the right, where the cycle control elements are drawn in gray. Within each loop, the invoked function vul is tested, and if and only if the taint bytes 180-187 / 367-370 are mutated, the verification of wrapper is bypassed.

The generated log contains the recorded exception information of the second round:

Listing 4 The log of triggered exception

```
......
403285: TaintByte bb/172 at 243a82 on 2-1
......
403285: Mutate 243a82 from 00 to ff
Exception c000001d on 000c0b0a @0
EAX:12                EBX:7ffd3000
ECX:0                 EDX:12
ESI:0                 EDI:0
ESP:24396b            EBP:44414548
roll back
```

According to the log, within the second layer of the taint function stack, taint byte 0xbb/0x172 was detected at memory position 0x243a82. Subsequently, according to the inner-to-outer mutation strategy, the header spec-

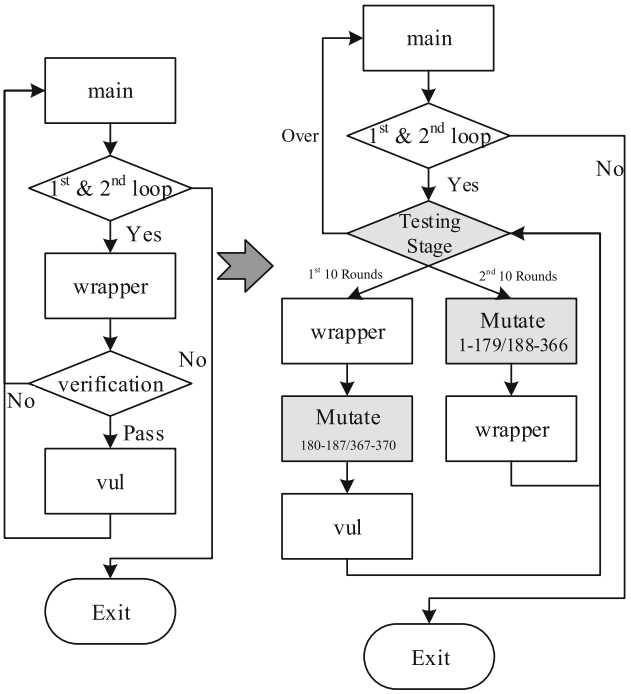

Fig. 7 Fuzz testing procedure

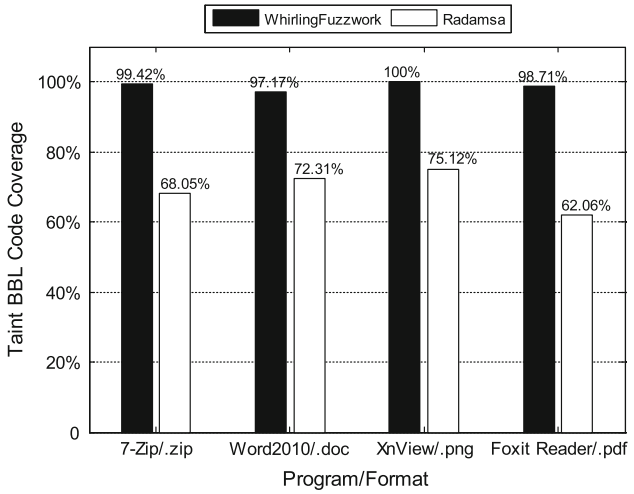

Fig. 8 Taint basic block coverage

ifier is bypassed, and the payload data are mutated. After changing memory byte `0x243a82` from `0x00` to `0xff`, an exception `0xc000001d` is triggered, implying that an invalid instruction is met, where the instruction address is `0x000c0b0a` as indicated by `EIP`, and the `EBP` is changed to `0x44414548`. It is easily deduced that once the `0xbb` byte of the text is changed to a non-0 value and when the function `vul` is returned, the `header` specifier of the second part of the content will overwrite the restored `EBP` on the top of the stack, and the following DWORD further overwrites the return address. At this stage, it is demonstrated that the fuzzing procedure managed to bypass the initial verification stage and directly mutate the tainted memory data to trigger an exception and that the corresponding sample is correctly recorded to re-trigger the same exception.

As a contrast, we tested the target program with Peach Fuzzer (Michael and Seth 2014), a widely used file and protocol fuzzing tool, under its random strategy. Because mutating only two out of the 370 bytes of the file could trigger the exception, a randomized mutation is difficult to successfully locate it. After fuzz testing for 500 rounds, the result showed that only two rounds triggered an exception, and one of them is retested as NonReproducable, which mutated the 370th byte and the exception depends on the raw data following the stack in the memory. This supports the judgment the WhirlingFuzzwork performs better than traditional fuzzing tools at least in the scope of bypassing program verifications and checksums.

4.2 Effectiveness

A feature of in-memory fuzz testing on function granularity is that, unlike normal fuzz testing, it is possible to traverse the finite conjunctions of execution paths within a function call stack. This feature indicates that this technique could technically reach a deeper invocation depth and provide larger code coverage.

To compare the code coverage of WhirlingFuzzwork to those based on external file mutation, we construct a group of contrast experiments. Given a normal sample input file, assume that the set of functions process the file data through a taint analysis. Subsequently, with instrumentation and inspection, calculate the coverage of these taint functions during the execution of the in-memory fuzz testing procedure. For comparison, generate a set of 100 mutated samples with a traditional test case generator and evaluate the average taint basic block (BBL) code coverage when processing these samples. We selected the generator developed by Oulu University, Radamsa (Aohelin 2010), which is widely used for file fuzz testing. The results obtained from the 4 selected file types are listed in Fig. 8.

According to the results, the taint code coverage lies between 60 and 75 % when processing randomly mutated cases, which depends on the ratio of the effective payload to the total file content. In contrast, it is guaranteed that each recognized taint function is executed during the in-memory fuzz testing, which yields a basic-block coverage of 95 % or greater.

4.3 Efficiency

Compared to existing fuzz testings, which need to restart the target process to undertake a single round of testing, in-memory fuzz testing is a one-process procedure: in each

round, only a certain function chain needs to be replayed in its execution context; therefore, the overhead for each testing round is technically greatly reduced.

To investigate the testing efficiency, we selected the widely used Peach Fuzzer (Michael and Seth 2014) version 3.1.124 for comparison. Without a loss of generality, we selected two experimental targets: a console program, dvips.exe, which represents programs with a high ratio of file data processing workload, and the GUI program Microsoft Paint mspaint.exe, which represents programs with multi-threads and sparse file-processing operations.

4.3.1 dvips Testing

dvips is a component program of the MiKTeX TEX compiling kit, which is used to convert a device independent file format (DVI) into a PostScript file. The source dvi files are generated from a LATEX file with a set of structural elements, and the complexity of the conversion depends on the diversity of the dvi structure hence the LATEX source file implicitly.

As the first step, we selected a LATEX file with the most elemental components and compile it to get a 216-byte dvi file. After executing dvips command, this file is compiled into a 22,362-byte ps file. Set the number of testing rounds against each single executed taint function to be 5. As a result, the total testing procedure required 6895 rounds of loops. Having experimented with Peach Fuzzer in its random mutation strategy for the same amount of testing cycles, the time consumption of both tools are shown in the curves in Fig. 9.

As indicated in Fig. 9, to conduct the same amount of 6800 rounds of tests, Peach Fuzzer consumed 3979 s in all with a steady execution rate of about 585.15 ms/round. In contrast, WhirlingFuzzwork consumed 290 s and the average execution rate is about 42.65 ms/round, with a 92.7 % enhancement in speed compared to the former.

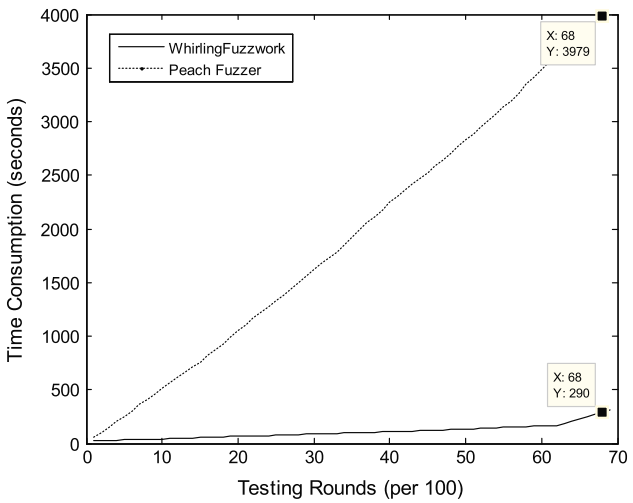

Fig. 9 Testing time against dvips

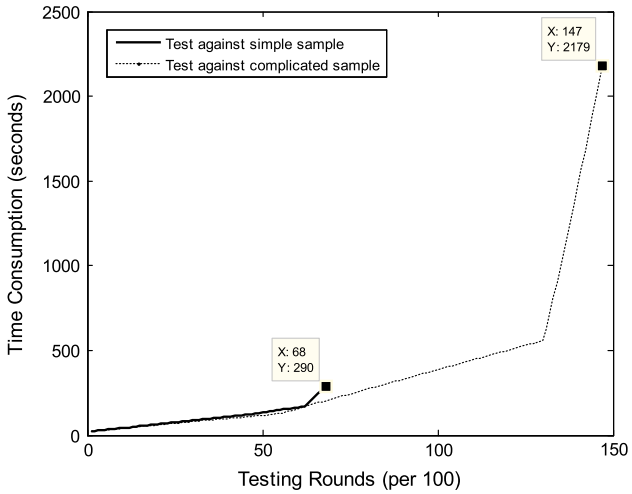

Fig. 10 Testing time against dvips

However, it was observed during the experiment that most of the randomly mutated sample dvi files by Peach failed the verification of dvi structure in dvips, which led to early quitting of the process, whereas approximately 1 out of 100 mutated dvi files reached the conversion step with about 3 s processing. This indicated that if the Peach Fuzzer is inducted with a proper dvi format specification and less violating the basic structure of the sample files, its time consumption would largely increased.

It was also observed that, unlike Peach, the testing rate of WhirlingFuzzwork was not consistent along the whole time; Around the 6200th testing cycle, there was an obvious slowdown of rate. This reflected that the rate is controlled by the complexity of the taint call stack under testing. Another way to observe the difference of rates in processing complexity is to set a contrast sample with higher complication. Therefore, we constructed another LATEX source file with more format codes such as customized fonts and font attributes, multiple pages, included graphics and tables, etc. The compiled dvi file is 1080-byte large. After being taint-analyzed, there were 32 functions within dvips.exe executive being involved in the converting task, which is the same as the set of processing the former simple sample. The time consumption of both experiments are illustrated in Fig. 10.

As shown in Fig. 10, to fuzz the procedure of processing the more complicated dvi sample file, the testing undertook 14,755 cycles in all. During the prior 6500 testing rounds, the execution rate is approximately the same as the less complicated one. As the experiment continued, there was a notable slowdown of rate around the 13,000th round. This is also in correspondence with the observation of the slowdown of the former experiment. It could be concluded from the fact, that the execution rate of WhirlingFuzzwork depends on the average complexity of the taint function call stack of the tested program. Also, we could exclude some outer wrapping func-

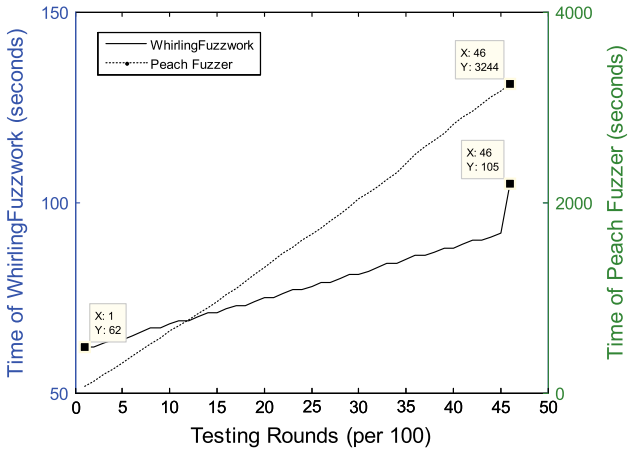

Fig. 11 Testing time against MS paint

tions, which do not make pivotal operations on the taint data, from the taint function list, so that the fuzzing efficiency could be largely enhanced.

4.3.2 MS paint testing

In this experiment, we selected a 1662-byte BMP image file to be parsed and rendered by Paint. To maximize the testing efficiency, set the number of testing rounds against each single executed taint function to be 5. According to the taint analysis, Paint loads the Windows system library `WindowsCodecs.dll` to decode the Bitmap data and renders the image with library GdiPlus.dll. In order to fuzz the data parsing logic, the WhirlingFuzzwork will target the functions in the former library only. The total number of tests is 4655, and the time consumption is shown in Fig. 11.

Peach Fuzzer automatically terminates the process under testing when its CPU utilization reaches 0. It required 3244 s to complete 4655 rounds, which is 696.9 ms/round. In contrast, WhirlingFuzzwork required 105 s overall, and there was an initialization period of 62 s from the start of the process until the execution of the first taint function. Therefore, the total time for all 4655 rounds is 43 s and 9.24 ms/round on average, which is 98.7 % faster than Peach Fuzzer.

5 Related work

Fuzz testing are a group of brute-force approaches for software quality assurance. According to the input data to be *fuzzed*, there are file format fuzzers (Sutton and Greene 2005) and network protocol fuzzers (Aitel 2002). Despite the differences in their targets and detailed implementations, all existing fuzzers take an *input-and-monitor* form of architecture, that the fuzzing payload data being generated or mutated externally and then fed into the target program, while a debugger agent waits to capture any exception; along the fuzzing course, the target process stays uninterfered and uninterrupted. In this way, any program exception caused by a malformed input sample can be directly back-played, but the chance of running into one is rather low because of the blindness of fuzzing technique. Such a technique regards the target program as a complete blackbox whose inner side cannot be seen, and the testing result dose not depend on any knowledge about the target. However, such uncertainty brings about high redundancy and low effectiveness, so that fuzz testing has always been regarded as the last chance to find any bugs in a program.

The concept of in-memory fuzzing was raised to solve a problem of traditional fuzzers, that in some cases the input cannot be produced or repeatedly fed into a restarted target process. As its initial introduction goes (Sutton et al. 2007), an in-memory fuzzer may take two kinds of architecture designs: loop-based one, with manually inserted looping codes into the original binary; and snapshot-based one, with a virtual machine basis and snapshots enabled. As indicated in the introduction, however, such implementations need either program source codes or strong low-level knowledge and handling of the system and process, which set a solid barrier for implementing a light-weight prototype.

As far as we could investigate, there is only one prototype in-memory fuzzer made public by the Corelan Team (2010). In their thorough introduction of the major techniques involved, a debugger PyDbg (Amini 2006) is employed to monitor the entries and exits of target functions to enforce cyclic execution on them, and whole-process memory layout is taken snapshot and restored, respectively. To locate the mutation objects, the prototype uses IDA (SA 2014) script to reverse engineer the target binary executable and export a complete list of inner functions and corresponding parameters, which are to be mutated while being invoked. However, such an implementation faces several drawbacks: no connection between external input and function memory data is established, so the mutations are just blind and need further manual analysis; the debugger-based execution scheduler is relatively low-speed and may simply fail to test all inner functions invoked by outer ones; the whole-process snapshot demands to export the data of the whole 2GB user memory space for a 32-bit process on each function entry, which is rather irrealizable; and any changes of system kernel objects in each loop are not properly handled or restored. In our implementation, these problems are solved with the techniques listed in Sect. 3 including taint information importing, instrumentation basis, save-on-written memory page snapshot and file object restoring, etc. The evaluation turns out to demonstrate the effectiveness of these novel techniques in the experiments scale.

As the fundamental of all the discussions about in-memory fuzzing technique, it is quite important to be able to deter-

ministically replay the execution of some certain part of process runtime. This has been of great interest of researchers in recent years, drawn designs and prototypes aiming to yield entirely accurate and light-weight replaying capability. Although such designs, as discussed in Sects. 2.3 and 2.4, may not match the needs for a highly available fuzzing framework because of their respective inherent features, they are not concrete obstacles for application as corresponding researches move forward. We would direct anyone who is interested in those researches to follow their progress, such as DrDebug (Wang et al. 2014) on debugging and Samsara (Ren et al. 2015) on virtualization. Besides, to start with binary instrumentation, there are also some interesting researches based on other platforms which could be referenced (Ryndin and Gaisaryan 2012; Bhansali et al. 2006).

6 Conclusion

In this article, we presented the concept of in-memory fuzz testing, which aims to eliminate the blindness of the existing fuzz methods. Furthermore, we presented the data flow information from a taint analysis to describe a taint-function-oriented API testing framework. We discussed the necessary technical components and key points in our own implemented prototype, WhirlingFuzzwork, and presented thorough experiments and results to demonstrate its availability and advantages in terms of testing efficiency.

Acknowledgements The research did not involve human participants or animals. The sources of funding include National Natural Science Foundation of China (No. 61272493). There are no potential conflicts of interests.

Compliance with ethical standards On behalf of, and having obtained permission from all the authors, I declare that: the material has not bee published in whole or in part elsewhere; the paper is not currently being considered for publication elsewhere; all authors have been personally and actively involved in substantive work leading to the report, and will hold themselves jointly and individually responsible for its content.

Conflict of interest The authors declare that they have no confilict of interest.

References

Aitel D (2002) The advantages of block-based protocol analysis for security testing. Immunity Inc, pp 105–106

Amini P (2006) Paimei-reverse engineering framework. In: RECON06: reverse engineering conference, Montreal, Canada

Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A (2003) Xen and the art of virtualization. ACM SIGOPS Op Syst Rev 37(5):164–177

Bellard F (2005) Qemu, a fast and portable dynamic translator. In: USENIX annual technical conference, FREENIX track, pp 41–46

Bhansali S, Chen WK, De Jong S, Edwards A, Murray R, Drinić M, Mihočka D, Chau J (2006) Framework for instruction-level tracing and analysis of program executions. In: Proceedings of the 2nd international conference on virtual execution environments, pp 154–163. ACM

Corelan Team (2010) In memory fuzzing. https://www.corelan.be/index.php/2010/10/20/in-memory-fuzzing/

Cui B, Wang F, Guo T, Dong G, Zhao B (2013) Flowwalker: a fast and precise off-line taint analysis framework. In: Emerging intelligent data and web technologies (EIDWT), 2013 fourth international conference on, pp 583–588. IEEE

Dunlap GW, Lucchetti DG, Fetterman MA, Chen PM (2008) Execution replay of multiprocessor virtual machines. In: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on virtual execution environments, pp 121–130. ACM

Hex-Rays SA (2014) Ida pro disassembler. https://www.hex-rays.com/products/ida/

Laadan O, Viennot N, Nieh J (2010) Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In: ACM SIGMETRICS performance evaluation review, vol 38, pp 155–166. ACM

Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K (2005) Pin: building customized program analysis tools with dynamic instrumentation. ACM Sigplan Notices 40(6):190–200

Michael E, Seth H (2014) Peach-cross-platform smart fuzzer. http://sourceforge.net/projects/peachfuzz/

Newsome J, Song D (2006) Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software

Patil H, Pereira C, Stallcup M, Lueck G, Cownie J (2010) Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In: Proceedings of the 8th annual IEEE/ACM international symposium on code generation and optimization, pp 2–11. ACM

Ren S, Li C, Tan L, Xiao Z (2015) Samsara: efficient deterministic replay with hardware virtualization extensions. In: Proceedings of the 6th Asia-Pacific workshop on systems, p 9. ACM

Radamsa A (2010) https://www.ee.oulu.fi/research/ouspg/Radamsa

Ryndin M, Gaisaryan SS (2012) Deterministic replay of program execution based on valgrind framework. In: Proceedings of the spring/summer young researchers colloquium on software engineering, p 6

Schwartz EJ, Avgerinos T, Brumley D (2010) All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: Security and privacy (SP), 2010 IEEE symposium on, pp 317–331. IEEE

Srinivasan SM, Kandula S, Andrews CR, Zhou Y (2004) Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In: USENIX annual technical conference, general track, pp 29–44. Boston, MA, USA

Sutton M, Greene A (2005) The art of file format fuzzing. In: Blackhat USA conference

Sutton M, Greene A, Amini P (2007) Fuzzing: brute force vulnerability discovery. Pearson Education

Wang T, Wei T, Gu G, Zou W (2010) Taintscope: a checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: Security and privacy (SP), 2010 IEEE symposium on, pp 497–512. IEEE

Wang Y, Patil H, Pereira C, Lueck G, Gupta R, Neamtiu L (2014) Drdebug: deterministic replay based cyclic debugging with dynamic slicing. In: Proceedings of annual IEEE/ACM international symposium on code generation and optimization, p 98. ACM