9-2012

Learning Fine-Grained Structured Input for Memory Corruption Detection

Lei ZHAO

Debin GAO
Singapore Management University, dbgao@smu.edu.sg

Lina WANG

DOI: https://doi.org/10.1007/978-3-642-33383-5_10

Learning Fine-Grained Structured Input for Memory Corruption Detection

Lei Zhao[1,2], Debin Gao[2], Lina Wang[1]

[1] Computer School of Wuhan University, Wuhan, China
[2] School of Information Systems, Singapore Management University, Singapore
zhaolei.whu@gmail.com, dbgao@smu.edu.sg, lnwang@whu.edu.cn

Abstract. Inputs to many application and server programs contain rich and consistent structural information. The propagation of such input in program execution could serve as accurate and reliable signatures for detecting memory corruptions. In this paper, we propose a novel approach to detect memory corruptions at the binary level. The basic insight is that different parts of an input are usually processed in different ways, e.g., by different instructions. Identifying individual parts in an input and learning the pattern in which they are processed is an attractive approach to detect memory corruptions. We propose a fine-grained dynamic taint analysis system to detect different fields in an input and monitor the propagation of these fields, and show that deviations from the execution pattern learned signal a memory corruption. We implement a prototype of our system and demonstrate its success in detecting a number of memory corruption attacks in the wild. In addition, we evaluate the overhead of our system and discuss its advantages over existing approaches and limitations.

Keywords: memory corruption, dynamic taint analysis

1 Introduction

Memory corruption exploits usually involve overwriting significant memory segments such as return addresses and function pointers [21]. Typical memory corruption exploits include control-hijacking attacks (e.g., [10]) and non-control data attacks (e.g., [8]). Despite having a long history, memory corruption exploits are still one of the biggest challenges to computer security [19].

Many techniques have been proposed to fight against memory corruption exploits, e.g., secure language [20, 14], bug detection [17], safe library [25], bounds checking [26], etc. Some of these techniques require access or even changes to the source code (e.g., [3, 7, 20, 14]), which might not be suitable when dealing with commercial off-the-shelf applications. Dynamic approaches which do not require source code of the program include canary-based techniques [10], probabilistic defenses [4], runtime enforcement [25], dynamic taint analysis [21], control flow integrity [2], etc. These binary-level techniques are powerful and efficient against many attacks, however they also suffer from some limitations [3]. For example,

dynamic tainting such as TaintCheck cannot detect non-control data attacks [21]. Pointer tainting [18] may result in a large number of false alarms because of legitimate use of input as pointers [23].

In this paper, we investigate how fine-grained taint analysis and propagation of program inputs could fight against memory corruptions. Program inputs usually contain rich structural information, which has been shown to be useful in a number of security applications [13, 16]. Intuitively, programs usually parse an input into various fields, which contain independent semantics and are subsequently processed by different instructions [6, 15]. A consistent and reliable pattern on the fine-grained structure of an input and the corresponding processing of it by different instructions could be used to capture normal execution of the program, since memory corruption exploits usually violate program semantics, e.g., the overflowed bytes are not processed by intended instructions. Thus the processing of exploits may not be consistent with that in benign executions. With this observation, memory corruptions could be detected by monitoring deviations of program execution from the pattern of input processing.

There are some difficulties in realizing such an intuition. First, the fine-grained structural information of the input might not be known. This could be due to a proprietary protocol used or lack of documentation. Moreover, even if a protocol description is available, it is usually not implementation specific, which might introduce noise to the detection (e.g., two independent fields in an input might be processed by the same instructions in a similar way in an optimized implementation). Second, it is unclear how to model the propagation and processing of the fine-grained structural input and to define patterns to catch the deviations. Third, existing dynamic tainting systems do not support the monitoring of fine-grained structural inputs.

We propose a novel technique called FiGi to monitor fine-grained input information for detecting memory corruptions. FiGi extends existing dynamic taint systems to enable precise monitoring of the propagation of individual input bytes (and their corresponding taint tags) during program executions. It learns and extracts structural information in program inputs by analyzing the execution context of every input byte. To normalize the input structures for specific inputs, FiGi uses a tree structure to model the input as well as its propagation in program executions, and constructs normal patterns by monitoring benign executions of the program. We implement FiGi and demonstrate its success in detecting a number of memory corruption exploits in the wild, include both control-hijacking and non-control data attacks. We additionally evaluate the overhead in using FiGi, and discuss its advantages over existing dynamic techniques in detecting memory corruption exploits as well as its limitations.

2 Related Work

The closest related work to FiGi is dynamic taint analysis which has been proposed to detect attacks, to diagnose vulnerabilities, and to generate attack signatures. TaintCheck [21] detects attacks that overwrite return addresses, function

pointers, format vulnerabilities, and in general control-related memory corruptions [8]. Pointer tainting detects non-control data attacks [18], but over-tainting causes a large number of false alarms [23]. FiGi differs from these existing dynamic tainting systems in the granularity of tainting in that FiGi assigns a different taint tag to each individual byte of the input and monitors the propagation of every taint tag. As we will show in Section 5, FiGi is able to detect both the control-hijacking attacks and non-control data attacks.

Clause et al. proposed tainting memory allocation to improve memory safety [9, 12]. Instead of tainting the input of the program, they taint the memory segments and the corresponding pointers for dynamic memory allocations on the heap and stack. However, a drawback of their approach is that compiler optimization may consolidate multiple memory regions (especially for local variables) into a single allocation request, which hides the granularity required for effective detection of memory corruptions. FiGi, on the other hand, does not suffer from this limitation because our tainting is fine-grained and not limited to memory allocations.

A key step in FiGi is to learn the structure of inputs, which is closely related to protocol reversing [6, 15]. FiGi employs a similar idea as these protocol and format reversing techniques. However, the different scenarios in which the recovered structure is used pose very different requirements on the discovery of the input structure. FiGi uses the input structure to detect memory corruptions exploits, which requires very fine-grained input structure to be learned; on the other hand, protocol and data structure reversing could benefit from grouping multiple fields of the input together as long as the execution context is similar [15].

3 A Motivating Example and the Challenges

In this section, we present a motivating example to demonstrate the idea of FiGi as well as the challenges involved.

```
232   char ifname[MAX_PATH_LEN]; /* input file name */
233   char ofname[MAX_PATH_LEN]; /* output file name */

756   if(make_ofname() != OK) return;
757   ifd = OPEN(ifname, ascii && !decompress, RW_USER);

1072  local int make_ofname() {
1076  strcpy(ofname, ifname); //memory corruption
```

Fig. 1. A motivating example of vulnerable gzip

Fig. 1 shows a vulnerable code segment of `gzip-1.2.4` with an unsafe string copy from `ifname` to `ofname`. If the length of the filename (part of the input)

is larger than 1024 bytes, `strcpy` will result in an overflow of the buffer for `ofname`. Note that FiGi works on the binary level. The source code and the corresponding memory layout of variables in Fig. 1 and Fig. 2 are presented for clearer explanation only. FiGi does not need to know anything about the high-level symbols such as `ifname` and `ofname`.

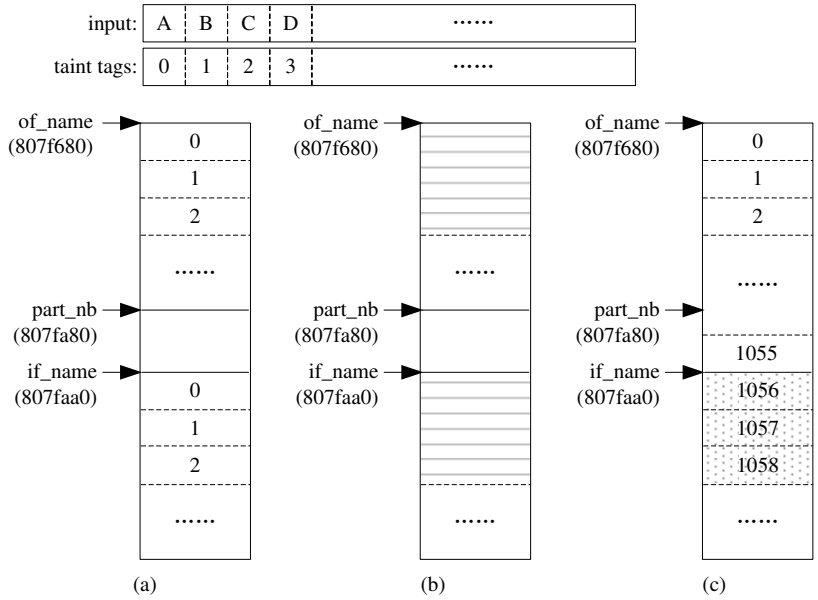


Fig. 2. Memory layout of gzip. `0x807f680`, `0x807fa80` and `0x807faa0` are the beginning memory addresses of `ifname`, `part_nb` and `ofname`, respectively.

We first examine what FiGi could learn from the fine-grained structural input in benign executions of `gzip`, when each byte of the command line input to `gzip` is assigned a different taint tag. Fig. 2(a) shows a portion of the memory layout in one particular execution of `gzip`, and we can see that each byte on the stack comes with a unique taint tag. At a closer look into the instructions processing the tainted input, we also realize that the tainted input bytes are stored into two buffers of which the memory addresses start from `0x807f680` and `0x807faa0`, respectively. Moreover, the bytes are all processed by the same instructions (`strcpy` and `open`) continuously during program executions. That is, the tainted input is processed as one unit. Therefore, FiGi could derive that the entire input contains only one field (see Fig. 2(b)).

To see the advantage of monitoring the fine-grained structural input, we consider a typical memory corruption exploit with inputs of 1200 bytes long at offset 0–1199 as shown in Fig. 2(c). When the input is longer than 1056 bytes, the unsafe string copy will overwrite segments of `part_nb` (the memory

address starts from `0x807fa80`) and `ifname` (the memory address starts from `0x807faa0`). Bytes with tags 1056–1199 would be accessed when `open` executes. FiGi would realize that the input portion with offset 1056–1199 is accessed by two instructions, namely `open` and `strcpy`, while the input portion with offset 0–1055 is accessed by `strcpy` only. Given the assumption that the input fields contain independent semantic and are subsequently processed by different instructions, from the exploit execution, the two portions with offset 1056–1199 and 0–1055 form two independent fields. This constitutes a deviation from what FiGi had learned from training of `gzip`, and triggers an alarm. Note that this deviation could not have been detected without a fine-grained monitoring of the structural input. Also note that such a memory corruption cannot be detected by TaintCheck [21] because tainted data does not change any function pointers or format characters.

Although this motivating example shows some intuition as to how the fine-grained structural input and its propagation during program executions could help detecting memory corruptions, there are some challenges we face.

1. The structural information of the input might not be known, either due to a proprietary protocol in use or lack of documentation. FiGi assumes that the only information available is the binary program as well as some training inputs. Therefore, we need to design FiGi in such a way that the structural information is learned via training.
2. It is unclear how to best model the propagation and processing of the fine-grained structural input to catch memory corruptions and to minimize false alarms. As we shall explain in the next section, FiGi keeps multiple patterns for each program to minimize false alarms.

4 System Design

4.1 Patterns and Deviations

We design two types of patterns in FiGi based on the observations that 1) the input structure is well-defined and consistent; and 2) fields accessed by different instructions are independent of one another. With these two observations, we capture the independent fields and the input structures. At the same time, for each independent field, we capture the execution context in which the field is processed as well.

Definition 1. *An independent field of the input consists of several continuous bytes which are always accessed as one unit in the program execution.*

Definition 2. *The accessing location of an independent field is the program execution context accessing the bytes of this field, which include the current call stack as well as the accessing instructions.*

With the two patterns, a program execution processing one input could be represented as follows.

Definition 3. *An input processing is denoted as* $R(i) = \langle S, F, E \rangle$*. In this formula,* $R(i)$ *refers to the program execution with the input* i*.* S *refers to the structure of* i*.* $F = \{f_1, f_2, ..., f_n\}$ *refers to the set of fields.* $E = \{e_1, e_2, ..., e_m\}$ *refers to the set of accessing locations.*

The input structure is specific with i (e.g., the offset interval of fields). With this impact, the field sets cannot be directly compared, and a normalized representation is required to generally represent the structured input. We will demonstrate this part in Sec. 4.5. Beside, there are mapping relationships between F and E. f_i is mapped with with more than one e_j, which indicates that one field could be accessed by several execution contexts, respectively.

For an unknown execution $R(i)$, the deviation is detected if 1) S is not matched with any of benign inputs, or 2) even S is matched, but the e_j of f_i cannot be matched. The first condition makes sense because the abnormal execution violates the input structures due to the misuse of corrupting bytes by unintended instructions. However, it is possible that the exploit could break and interrupt the input parsing, especially when the program is control hijacked and jumps to illegal instructions. These cases are common in multiple vulnerable programs (e.g, `AT-Tftp`, `ghttpd` in Sec. 5). As a result, we could only get a partial structure and the difference of structures may not be clear. In such cases, the second condition makes sense because a specific fields can only be accessed in several specific accessing locations. In Sec. 5.2, we will use the `ghttpd` program to illustrate.

4.2 Overview

As demonstrated in Sec. 3, we cannot expect the structural information of inputs is known, and we need to extract and learn patterns on input processing from benign executions. To overcome these challenges, the basic idea of FiGi is to model the propagation and processing of the fine-grained structured input through dynamic execution monitoring. Fig. 3 shows an overview of FiGi.

1. We adopt the dynamic taint analysis to dynamically monitor the program executions as well as the propagation of inputs. In addition, we extend the dynamic taint analysis to enable precise monitoring of the propagation of individual input bytes (and their corresponding taint tags), such fine-grained propagation could be used to capture the structured input.
2. To model the input processing and learn some patterns, we identify fields of the input via execution context comparison. This approach is similar with the protocol/format reversing techniques [15], except the difference that the field identification is performed online for the purpose of emergency responses to memory corruption attacks. Moreover, for exploits, the input structure may not be sufficient for deviation detection, because the memory corruption attack could break and interrupt the input parsing, from which the deviation cannot be observed and lead to false negatives. For effectiveness enhancement, we also collect the corresponding processing contexts for each

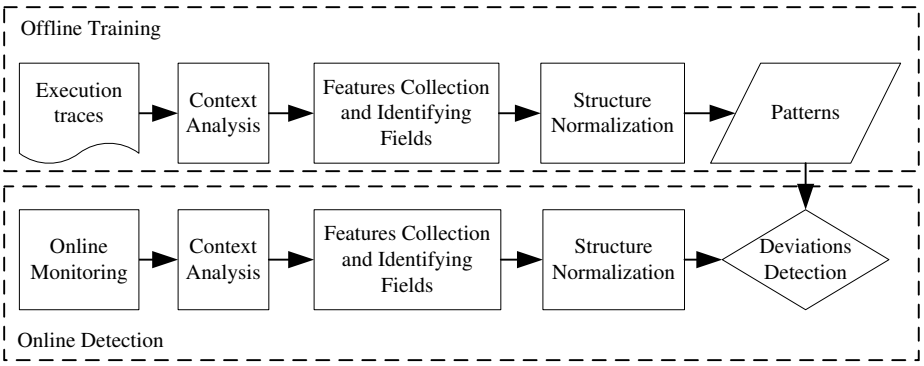


Fig. 3. The overview framework of our model

independent field based on the intuition that one field could be accessed in several specific execution contexts.

3. The identified fields are represented by the taint tags of a specific input. Since some fields are not length-fixed and some fields are optional, we need a normalized representation of structural inputs to compare the structures and figure out the deviations. We design to use the tree structure to model the structured input based on the observation that the input is processed subsequently and the input parsing looks like the construction of a tree.
4. By analyzing the execution context and identifying fields of benign executions, we generate patterns of the input processing and propagation, and use them to detect the deviation of executions with unknown inputs. The deviation detection is performed online. In details, we dynamically analyze the execution context and the propagation of fine-grained structured inputs to identify fields and collect patterns, whenever a field is identified, update $R(i)$ and perform deviation detection on $R(i)$ with trained ones.

4.3 Execution Monitoring and Context Tracking

To monitor the propagation of inputs and precisely capture the execution context, we extend the dynamic tainting [21, 6] to give each byte a unique taint tag. The taint tag includes two items, the taint source and the taint offset, which refer to the source of the input and the byte offset, respectively. Among the taint propagation, we make the destination operand has the union taint record for multiple source operands. For example, suppose the instruction is `ADD %eax, %ecx`. The taint records of `%eax` and `%ecx` are $\{1001, 3\}$ and $\{1001, 7\}$, respectively. After the execution, the taint record of `%ecx` will be $\{\{1001, 3\}\{1001, 7\}\}$.

During execution monitoring, we record two types of execution context information: the run-time call stack and the address of instructions that access tainted bytes [15]. To acquire the run-time call stack, we monitor the the function call and return instructions, as well as the stack frame balancing. For each

taint byte, we make every tainted instructions as well as the call stack to form an execution sequence of this taint byte. Then we compare the execution sequences of taint bytes with continuous offsets, if the execution sequences could be matched, we regard the continuous bytes belong to the same field.

4.4 Identifying Fields and Collecting Patterns

Algorithm 1: Identifying Fields

```
   Data: InstAddr(t_b): the instruction address accessing the taint byte t_b
   Data: ExecSeq[index]: the data structure storing the execution sequences
   Data: ExecSeq[index][t_b]: the execution sequence of the taint byte t_b
   Result: ⟨ offset intervals, accessing locations ⟩
 1 while instruction i do
 2     if i is tainted then
 3         for taint byte t_b in i do
 4             Insert InstAddr(t_b) into the ExecSeq[index][t_b];
 5         end
 6     end
 7     Call Stack Analysis;
 8     if call stack changes and a function returns then
 9         ContextComparison(ExecSeq[index]);
           // compare the execution sequences and identify fields
10         delete ExecSeq[index];
11         index−−;
12     end
13     if call stack changes and a function starts then
14         index++;
15         ExecSeq[index] = new ExecSeq;
16     end
17 end
```

In the dynamic protocol reversing techniques, the execution context analysis are performed off-line [6] and the inputs are assumed benign [6, 15]. For our problem scope, we cannot wait for the program to exit and should sponsor quick response to the anomaly as early as possible. That is, the execution context comparison and identifying fields should be performed online.

To identify fields online, an intuitive approach is to compare the execution context of a captured taint byte with those of its continuous bytes whenever the taint byte is captured. However, the taint bytes are dynamically processed and the order of accessed taint bytes is an undecided problem (e.g., the program could access another field such as a separator between the period of accessing two continuous bytes). To overcome the problem, we design to perform the context comparison whenever a function returns that causes the call stack changes. For

every calling of a function, we allocate a data structure to store the execution sequences of taint bytes which are accessed within the current call stack. When a function returns, we compare the execution context for the bytes that are only processed in this function, and the offset interval could be identified. The algorithm is shown in Algorithm 1.

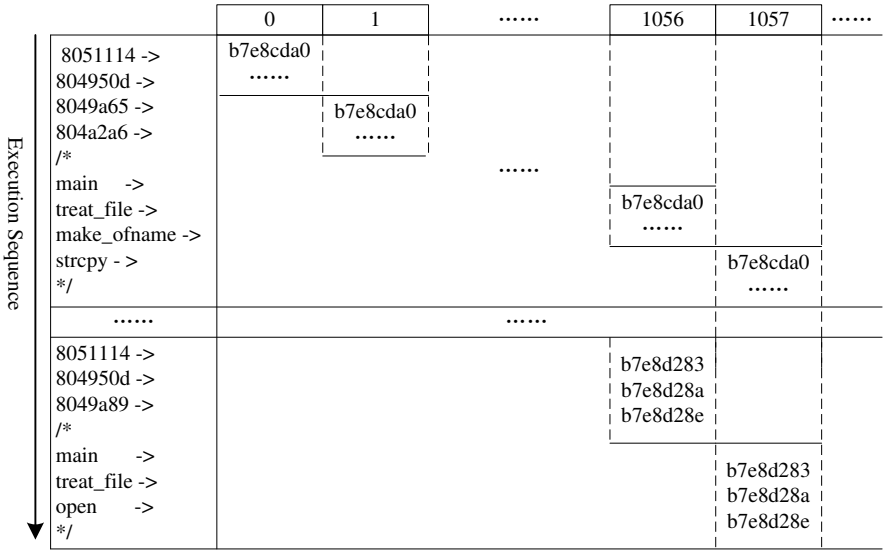


Fig. 4. The execution trace of the vulnerable gzip. The header refers to taint tags of input bytes, the left column refers to the call stacks, and the hex numbers refer to the address of instructions accessing the taint bytes.

This scheme leads to little impact on the effectiveness of identified fields but a little impact on the detection of memory corruptions. First, it is rarely that several parts of one field are processed with different functions. Second, memory corruption attacks could only be detected when the call stack changes, which causes a little delay.

Whenever a field is identified, we record the current call stack as well as the instruction address accessing this field, and make them as the accessing location of this field. Note that the field could be accessed by several instructions, we only regard the current call stack and the beginning instruction address as the accessing locations.

We still take the `gzip` example for detailed illustration. Fig. 4 shows a segment of execution contexts. When the program calls `strcpy`, we create a new object of data structure to store the execution sequences for every taint byte. Among the instructions of `strcpy`, we record the tainted instructions, and insert the instructions into the execution sequences of taint bytes. When `strcpy` returns, we find that the taint bytes share the same execution context and then

are grouped as one unit in `strcpy`. The field with the offset interval [0, 1199] (the length of the exploit is 1200) is identified. Another call stack is captured when program calls `open`. The bytes from offset 1056 to 1199 are accessed by the instructions of which the addresses are `b7e8d283`, `b7e8d28a`, and `b7e8d28e`. But the bytes from offset 0 to 1055 are not accessed in this function. As a result, a new field of which the offset interval is [1056, 1199] is identified.

For the two fields identified in this trace segment, [0, 1199] and [1056, 1199], the accessing locations identifying these two fields are shown in Fig. 5. The two fields are accessed by several instructions, we only record the beginning instruction addresses as the accessing locations.

offset interval	execution context
0-1199	8051114 -> 804950d -> 8049a65 -> 804a2a6 -> b7e8cda0 /* main -> treat_file -> make_ofname -> strcpy */
1056-1199	8051114 -> 804950d -> 8049a89 -> b7e8d283 /* main -> treat_file -> open */

Fig. 5. The accessing locations identifying the two fields

4.5 Structure Normalization

The identified fields are represented with the offset intervals. These offsets are specific to an input, because the values of some fields are user-defined and not every fields is length-fixed. For two inputs, the offset intervals are likely not identical and cannot directly to compare. We need to normalize the specific offsets with an abstract structure.

The tree structure is a well fit data structure to represent the input formats because the input could be a flatten or hierarchical structure [6, 15]. In our approach, we also employ the tree to normalize the structures of specific inputs, where nodes refer to the fields and edges refer to the flatten or hierarchical relationships.

During online monitoring and detection, the tree is dynamically built and initialized with a root node. Whenever a new field is identified, we search its parent node of which the offset interval is the smallest yet covers the offset interval of the new field, and then insert a new node into the tree as a child node of its parent node. If no parent is found, we will make the root node as the parent node. At the same time, for a parent node has several child nodes, we order these child nodes with a increasing order of their offset interval.

Table 1. Vulnerable Programs

Programs	Published Date	Vulnerable Description	Detected
3CTfpdSvc-0.11	2006-05	stack overflow	√
AT-Tftp-1.9	2008-05	stack overflow	√
knftpd-1.0.0	2011-10	stack overflow	√
tftpd32-2.21	2010-09	format string	√
nginx-0.6.38	2010-08	heap overflow	√
wu-ftpd-2.6.0	2001-01	format string (non-control)	√
ghttpd-1.4.3	2002-10	stack overflow (non-control)	√
floatFTP	2011-09	stack overflow (ROP attack)	√
gzip-1.2.4	2002-06	stack overflow (DoS)	√

5 Implementation and Evaluation

We implement FiGi on the platform of `Bitblaze` [24]. `TEMU`, the taint tracking component of `Bitblaze`, develops sophisticated data structures on the QEMU and could be very slow with an overhead at several hundreds [24].

`Bitblaze` uses shadow memory and a data structure of taint record to represent the taint tags for every bytes. FiGi extends the data structure of such taint record to an array to store the multiple tags, and modifies the taint propagation for instructions with multiple source operands. Compared with the original `Bitblaze`, the performance of FiGi is impacted by a litter larger memory movement of taint records and the multiple taint propagation for instructions that have multiple tainted source operands. FiGi results in the similar overhead as `Bitblaze`.

The off-line training is a stand-alone program that is performed on the execution traces collected from FiGi. The online monitoring, context analysis, and deviation detection are implement as a plugin of `TEMU`. Both the off-line training and online monitoring are implemented in C++ with about 3000 lines of code.

5.1 Evaluation on Attacks

We select several vulnerable programs to evaluate the effectiveness of FiGi. The selected programs are shown in Table 1. The exploits of vulnerable programs cover memory corruptions on stacks, format strings, and heaps. The execution of such exploits could result in both control-hijacking and non-control data attacks. In addition, we also select an attack exploit based on the Return-Oriented Programming (ROP), which is a research focus during recent years, and a DoS attack of which the memory corruption could lead the program to crash but hard to exploit.

As shown in Table 1, FiGi could detect all these memory corruptions. In general, memory corruptions are the root-cause of control-hijacking, non-control data attacks, and many memory errors. The approach behind FiGi is to detect

the misuse of user inputs by unintended instructions. Therefore, we claim that FiGi is transparent to the type of attacks, no matter the memory corruption is used for control-hijacking, non-control data attacks, or just deny of service.

5.2 A Case Study

We use an example to show a case study on how FiGi works to detect both the control-hijacking and non-control data attacks. There is a stack overflow vulnerability in the `log()` function in `ghttpd-1.4.3`. This vulnerability could be triggered when the `GET` package contains too many bytes. In [8], Chen et al. proposed a non-control data attack to overwrite a significant pointer and force the program to execute the "*\bin\sh*". We use both control-hijacking exploit and non-control exploit to compromise this program.

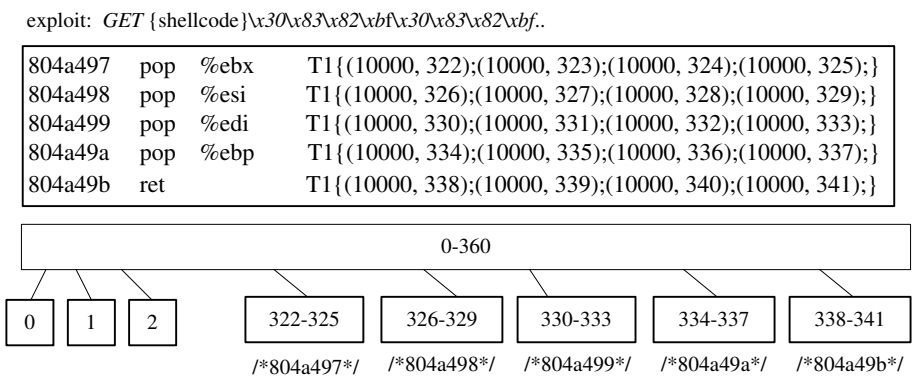


Fig. 6. The detection of the control-hijacking attack on ghttpd. `T1` refers to tainted, and `T0` refers to non-tainted. The taint tags of individual byte are separated with ";". Each taint tag is represented as `(source, offset)`, where `source` refers to the input source and `offset` refers to the offsets of every bytes.

Let us first examine the control-hijacking attack shown in Figure 6. We observe that operands of these 5 instructions are tainted. By comparing execution contexts, 5 fields are identified ([322, 325], [326, 329], [330, 333], [334, 337], and [338, 341]), of which the accessing locations are `804a497`, `804a498`, `804a499`, `804a49a`, `804a49b`. After the normalization, we compare the pattern of the exploit with trained ones and find that the accessing locations of the 5 fields are never present in the benign executions.

During the non-control attack, as shown in Figure 7, the exploit only overwrites the value of `ebx` and `esi`. FiGi detects that 2 fields are identified and their accessing locations are not present in benign executions.

Note that the execution of `log()` function is much earlier before the `ghttpd` parsing the entire `GET` request package, thus the tree structure at this moment is simple. However, the tree structure of a benign input is much more complicate,

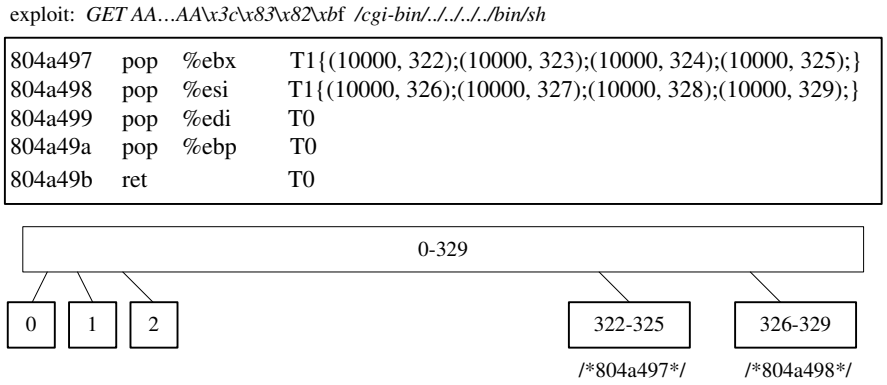


Fig. 7. The detection of the non-control data attack on ghttpd.

and a similar result could be seen in [15]. There is no difference between this partial tree structure and a benign tree because this partial tree is a sub-tree of that. The deviation in this example is detected by matching the accessing locations.

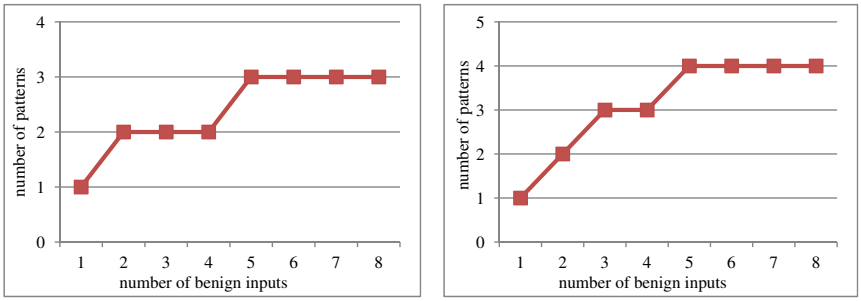
5.3 False Negative and False Positive

In our experiments, we encounter no false negatives. It means that we missed no attacks among these control-hijacking, non-control data attacks, and memory corruption errors. Among these attacks, all the exploits try to overwrite some significant data structures, including the internal data structures (such as the return addresses) and significant variables. After the corruptions, some bytes would further be misused by unintended instructions, which may lead to the deviation of input structures or the deviation of execution context for a specific field of user input. These anomalies could be captured by the patterns of input processing. However, there is no guarantee that the false negatives will not occur.

The false positives is a big challenge in FiGi. The main reason to arise false positive is the coverage. That is, if the coverage is not sufficient, then we could miss some input processing patterns. For example, record sequences [11] are common in some user inputs such as images, videos and others. Generally, the processing of such sequences will be a loop. In such cases, the number of loop paths is large and hard to be full covered, and FiGi may generate false positives for benign executions covering untrained paths.

In addition, there is no close relationships between the input structure and the execution paths. Inputs with the same structure but different values of fields could also generate different patterns. This scenario is not rare because some significant input values could affect the program behaviors. We are interested in the training overhead of inputs with identical structures. In the two case studies, we only change the value of some fields but keep the structure unchanged, and then construct 8 benign inputs for `gzip` and `nginx`, respectively. The training

overhead is shown in Fig. 8, and the numbers of trained patterns are 3 and 4, respectively.



(a) gzip (b) nginx

Fig. 8. The training overhead of the gzip and nginx

5.4 Comparison with Peer Techniques

We perform a theoretical comparison between FiGi and peer techniques on their abilities to detect attacks. Since FiGi works on the binary level, we just select peer techniques works on binaries (e.g., TaintCheek [21], PointerTainting [18], CFI [2], Clause2007 [9, 12]) and transparent protection systems (e.g., StackGuard [10], Data Execution Protection (DEP for short) [1], ALSR [4]).

Table 2. Comparison with Peer Techniques

	Attacks								Performance	Errors
	S(C)	F(C)	H(C)	F(N)	S(N)	H(N)	ROP	DoS		
StackGuard [10]	√	•	•	×	×	×	•	×	≈ 1x	FN
DEP [1]	√	√	√	×	×	×	×	×	≈ 1x	FN
ALSR [4]	√	√	√	√	√	√	•	•	≈ 1x	FN
TaintCheck [21]	√	√	√	×	×	×	√	×	1.5x-30x	FN
PointerTaint [18]	√	√	√	√	√	√	√	×	1.5x-30x	FP/FN
CFI [2]	√	√	√	×	×	×	√	×	1x-2x	FN
Clause2007 [9, 12]	√	√	√	√	√	√	√	•	1x-500x	FN
FiGi	√	√	√	√	√	√	√	√	100x-1000x	FP

In Table 2, we list the control-hijacking and non-control data attacks which may be caused by stack overflow, format string, and heap overflow. For short

representation, we use S, F, H to denote the stack overflow, format string vulnerability, and heap overflow, respectively. We use C and N to denote the control-hijacking and non-control data attacks, respectively. The symbol S(C) refers to the control-hijacking attacks caused by stack overflow. Other symbols could be similarly explained. In addition, we also select ROP and DoS memory corruptions as comparison features. ROP could be caused by stack overflow, format string, and heap overflow, and in Table 2, we just use ROP to represent attacks passing DEP.

In Table 2, we use $\surd$, $\times$, $\bullet$ to denote that the attack can be detected, cannot be detected, and uncertain. $\bullet$ means the protection mechanism may have variants, which could be the improvement of previous techniques, or a different implementation. Attacks could be detected by some of them, but may not be detected by other variants. For example, StackGuard [10] places a hard-to-predict canary before the return address on the stack, which could detect stack smashing but cannot detect control-hijacking attacks caused by format string and heap overflow. As an improvement, StackShield protects the stack by copying the return address to a "secure" location. Some ASLR [4] only randomize the load memory but keep the relative addresses un-randomized, than it cannot detect the memory corruptions between variables (e.g., wu-tftp and gzip). As an improvement, some fine-grained ASLR could also randomize the relative addresses (e.g., ASLR through binary transformation [4]), which could mitigate more attacks. As we discussed in Sec. 2, Clause2007 [9, 12] may lose its ability for non-control memory corruptions if the symbol table is unavailable.

FiGi could detect all these memory corruptions based the consistent and reliable foundation that memory corruption exploits violate the program semantics, leading some input bytes being misused by unintended instructions, and the anomaly could be captured through patterns on the input processing. We encounter no false negatives (FN for short in Table 2), but other peer techniques could result in false negatives more or less. It means that FiGi misses the least number of attacks. However, FiGi could result in false positives (FP for short in Table 2).

From Table 2, we could also observe that performance is the shortage of FiGi. The transparent protections are very fast. The prototype of TaintCheck [21] could result in the overhead about 30x. Recent advances on dynamic taint analysis could reduce the overhead to 1.5x [5]. There is no quantified performance in PointerTainting [18], but it should has the similar overhead with dynamic taint analysis. The overhead of CFI is about 1x-2x. The Clause2007 [9, 12] prototype based on software emulation could cause the overhead of 100x-500x, and the improved overhead with hardware-assistant is about 1x-2x. We build FiGi on TEMU [24], of which the overhead is more than several hundreds. To improve the performance, several optimizations could be used, such as simplifying taint tracking instructions [22], designing novel memory layout to reduce the overhead caused by taint propagation [5], faster emulator [5], and hardware assistant [12]. Although these optimizations may not be directly used, we could learn from these and design similar schemes.

6 Limitations and Discussions

In this paper, the fields mainly include semantic independent units, and we do not recognize the higher level data types. For the complex and higher level data types such as structs and unions, every item in the struct or union is treated as one independent field, if each item is processed as one unit among the execution.

Comparing the execution contexts could lead to over-fine granularity [15]. For example, `strcmp` could return after accessing the first few bytes. In such cases, the first few bytes will be regarded as one field and FiGi may divide the string into several parts. Fortunately, such over-fine granularity shall occur in all executions, both benign and abnormal ones. Therefore, this limitation bring little noise to the deviation detection.

7 Conclusion

In this paper, we propose a novel approach FiGi to detect memory corruptions at the binary level. FiGi identifies individual parts in an input and learns the pattern in which they are processed. We implement a prototype of FiGi and demonstrate its success in detecting a number of memory corruption attacks in the wild. The experiments shown that FiGi is effective to detect memory corruptions.

Acknowledgment

This work is partially supported by the National Natural Science Foundation of China (no. 60970114), the Doctoral Priority Development Projects granted by the Chinese Ministry of Education (no. 20110141130006), the Research Award for Excellent Doctoral Student granted by Chinese Ministry of Education, and the Northeast Asia Grant Program by SafeNet.

References

1. Data execution protection, `http://technet.microsoft.com/en-us/library/cc738483(WS.10).aspx`
2. Abadi, M., Budiu, M., Erlingsson, ., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Transactions on Information and System Security 13(1), 1–40 (2009)
3. Akritidis, P., Cadar, C., Raiciu, C., Costa, M., Castro, M.: Preventing Memory Error Exploits with WIT. In: 2008 IEEE Symposium on Security and Privacy. pp. 263–277 (2008)
4. Bhatkar, S., DuVarney, D.: Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In: Proceedings of USENIX Security (2003)
5. Bosman, E., Slowinska, A., Bos, H.: Minemu: The World's Fastest Taint Tracker. In: Proceedings of RAID (2011)

6. Caballero, J., Yin, H., Liang, Z., Song, D.: Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In: Proceedings of CCS. pp. 317–329 (2007)
7. Castro, M., Costa, M., Harris, T.: Securing software by enforcing data-flow integrity. In: Proceedings of OSDI. pp. 147–160 (2006)
8. Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-control-hijacking attacks are realistic threats. In: Proceedings of USENIX Security (2005)
9. Clause, J., Doudalis, I., Orso, A., Prvulovic, M.: Effective memory protection using dynamic tainting. In: Proceedings of ASE. pp. 284–292 (2007)
10. Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of USENIX Security (1998)
11. Cui, W., Peinado, M., Chen, K., Wang, H., Irun-Briz, L.: Tupni: Automatic reverse engineering of input formats. In: Proceedings of CCS. pp. 391–402 (2008)
12. Doudalis, I., Clause, J., Venkataramani, G., Prvulovic, M., Orso, A.: Effective and Efficient Memory Protection Using Dynamic Tainting. IEEE Transactions on Computers 61(1), 87–100 (2012)
13. Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. In: Proceedings of PLDI. vol. 43, pp. 206–215 (2008)
14. Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., Wang, Y.: Cyclone: A safe dialect of C. In: Proceedings of USENIX ATC. pp. 275–288 (2002)
15. Lin, Z., Jiang, X., Xu, D., Zhang, X.: Automatic protocol format reverse engineering through context-aware monitored execution. In: Proceedings of NDSS (2008)
16. Lin, Z., Zhang, X.: Reverse Engineering Input Syntactic Structure from Program Execution and Its Applications. IEEE Transactions on Software Engineering 36(5), 688–703 (2010)
17. Livshits, V.B., Lam, M.S.: Tracking pointers with path and context sensitivity for bug detection in C programs. In: Proceedings of FSE. vol. 28, pp. 317–326 (2003)
18. Nakka, N., Kalbarczyk, Z., Iyer, R.: Defeating Memory Corruption Attacks via Pointer Taintedness Detection. In: Proceedings of DSN. pp. 378–387 (2005)
19. National Institute of Standards and Technology: National vulnerability database statistics, `web.nvd.nist.gov/view/vuln/statistics`
20. Necula, G.C., McPeak, S., Weimer, W.: CCured: type-safe retrofitting of legacy code. In: Proceedings of POPL. pp. 128–139 (2002)
21. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proceedings of NDSS (2005)
22. Qin, F., Wang, C., Li, Z., Kim, H.s., Zhou, Y., Wu, Y.: LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In: Proceedings of Micro. pp. 135–148 (2006)
23. Slowinska, A., Bos, H.: Pointless tainting? Evaluating the practicality of pointer tainting. In: Proceedings of EuroSys. pp. 61–74 (2009)
24. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: A New Approach to Computer Security via Binary Analysis. In: Proceedings of the 4th International Conference on Information Systems Security (2008)
25. Tsai, T., Singh, N.: Libsafe: transparent system-wide protection against buffer overflow attacks. In: Proceedings of DSN. pp. 541–550 (2002)
26. Younan, Y., Philippaerts, P., Cavallaro, L., Sekar, R., Piessens, F., Joosen, W.: PAriCheck: an efficient pointer arithmetic checker for C programs. In: Proceedings of AsiaCCS. pp. 145–156 (2010)