Available online at www.sciencedirect.com

SciVerse ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Automatic and lightweight grammar generation for fuzz testing

Su Yong Kim [a,*], Sungdeok Cha [b], Doo-Hwan Bae [c]

[a] The Attached Institute of ETRI, 909, Jeonmin-dong, Yuseong-gu, Daejeon, South Korea
[b] Computer Science and Engineering Department, Korea University, Anam-dong Seongbuk-gu, Seoul 136-701, South Korea
[c] Division of Computer Science, EECS Department, Korea Advanced Institute of Science and Technology (KAIST), 335 Gwahak-ro, Yuseong-gu, Daejeon, South Korea

ARTICLE INFO

Article history:
Received 13 October 2012
Accepted 1 February 2013

Keywords:
Hybrid fuzz testing
Whitebox fuzz testing
Blackbox fuzz testing
Grammar-based fuzzer
ActiveX control

ABSTRACT

Blackbox fuzz testing can only test a small portion of code when rigorously checking the well-formedness of input values. To overcome this problem, blackbox fuzz testing is performed using a grammar that delineates the format information of input values. However, it is almost impossible to manually construct a grammar if the input specifications are not known. We propose an alternative technique: the automatic generation of fuzzing grammars using API-level concolic testing. API-level concolic testing collects constraints at the library function level rather than the instruction level. While API-level concolic testing may be less accurate than instruction-level concolic testing, it is highly useful for speedily generating fuzzing grammars that enhance code coverage for real-world programs. To verify the feasibility of the proposed concept, we implemented the system for generating ActiveX control fuzzing grammars, named YMIR. The experiment results showed that the YMIR system was capable of generating fuzzing grammars that can raise branch coverage for ActiveX control using highly-structured input string by 15–50%. In addition, the YMIR system discovered two new vulnerabilities revealed only when input values are well-formed. Automatic fuzzing grammar generation through API-level concolic testing is not restricted to the testing of ActiveX controls; it should also be applicable to other string processing program whose source code is unavailable.

1. Introduction

Blackbox fuzz testing is effective for finding vulnerabilities when the composition of the input value is simple rather than complex. In fact, many vulnerabilities found in the real world thus far could be triggered using input values with simple formats. However, when a program is designed to check the validity of complicated input string formats, blackbox fuzz testing can only be used to cover a severely limited code domain. To overcome this problem, blackbox fuzz testing is performed using a grammar that delineates the format information of input values (Protos; Aitel, 2002; Eddington; Amini and Portnoy). However, such studies were impeded by the feature that the necessary grammars had to be described manually (Gorbunov and Rosenbloom, 2010). It is virtually impossible to write a grammar manually if the format of the input value is unknown; even when the format is widely known, manually describing all the necessary grammars requires significant time and cost. In addition, test cases derived from specification alone may not reveal potential errors. If, for example, developers implemented customized (but optimized for performance) protocols, specification-based test cases may become inadequate.

* *Corresponding author*. Tel.: +82 42 870 2158; fax: +82 42 870 2222.
E-mail addresses: sweetlie@hotmail.com, sw22t1i2@gmail.com (S.Y. Kim), scha@korea.ac.kr (S. Cha), bae@se.kaist.ac.kr (D.-H. Bae).
0167-4048/$ – see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cose.2013.02.001

Similarly, test cases generated from the specification may result in excessive testing activity if developers decide to implement only partial features essential to the current requirements (Drewry and Ormandy, 2007). Our approach, however, can overcome such limitations because test cases are derived from implementation without being biased by specification.

Another alternative to blackbox fuzz testing is the use of concolic testing (Cadar et al., 2008a, 2008b; Godefroid et al., February 2008; Sen et al., 2005; Tillmann and de Halleux, 2008). Due to recent advancements in constraint-solving technology, it has become possible to apply concolic testing to real-world programs (Barrett and Berezin, 2004; Kieun et al., 2009). In particular, SAGE (Scalable, Automated, Guided Execution) is capable of carrying out dynamic test case generation on ×86 Windows binary programs. When part of the input file is being used as a condition for a branch, SAGE extracts the branch condition as a constraint. It then negates the extracted constraints one by one and carries out constraint solving in order to generate concrete test cases needed to execute a new path (Godefroid et al., 2012).

Concolic testing can greatly enhance code coverage if enough time is given. However, it should perform time-consuming instruction logging process and constraint solving which is an NP-complete problem (Kieun et al., 2009).

We suggest the new method to improve the code coverage of blackbox fuzz testing through ACT (API-level concolic testing). In general, solving the constraints obtained as a result of concolic testing yields a concrete test case. However, we use the constraints collected through ACT to generate fuzzing grammars rather than concrete test cases. Fuzzing grammars explicitly differentiate fields that are essential for executing a particular path from those that can be arbitrarily modified. Concrete test cases created by substituting fields that do not affect program paths with long strings, etc, can trigger various security bugs while covering a wide range of paths.

ACT collects as constraints the parameter and return values of well-known functions that can affect the flow of an application, and negates these constraints to generate a fuzzing grammar capable of executing a different path. This approach has no overhead for instruction logging and constraint solving although its applicability is limited to functions defined in widely used standard libraries. Developers generally prefer to use built-in functions (e.g., strcmp) rather than implementing their own functions with the same feature. Therefore, such restrictions have no adverse impact on applicability to real-world applications.

In this paper, an SFCF (string format checking function) is defined as a well-known library function that compares two separate strings or checks whether one string exists within another. As string comparison is usually performed in order to select a path for execution, we can presume that a program's execution path will change according to the return value of the SFCF. Therefore, different execution paths may be taken by altering the input string to negate the return value of the SFCF.

Our contributions are as follows:

First, we automatically generate fuzzing grammars to improve the code coverage of blackbox fuzz testing. Fuzzing grammars explicitly differentiate fields that affect paths from those that do not. Grammar-based fuzzers can use fuzzing grammars to derive concrete test cases by replacing fields that do not affect execution paths with long strings, etc. Such test cases would traverse a variety of different paths and can trigger security bugs.

Second, we propose ACT for the generation of fuzzing grammars. ACT collects constraints at the API level rather than at the instruction level. Thus, it can generate fuzzing grammars for increasing code coverage without going through the time-consuming step of instruction logging and constraint solving.

Third, four rules are proposed for transforming the results of an SFCF call into constraints. Specifically, SFCFs are divided into four groups by analyzing their semantics, and a method to transform the call results of each function group into corresponding constraints is proposed. Fuzzing grammars are generated by negating the collected constraints one by one and merging with the fuzzing grammar applied to the current test case. The time complexity of constraint solving is O(1) in our proposed technique.

We implemented the YMIR system to demonstrate effectiveness of the proposed technique using real-world applications such as ActiveX controls. To the best of our knowledge, it is the first tool designed to automate whitebox fuzz testing on ActiveX controls. It takes an ActiveX control as input and delivers fuzzing grammars as its output.

The experiment results show that fuzzing grammars can increase the code coverage for programs using complexly-formatted input strings by 15–50% in comparison to random fuzz testing. In addition, the YMIR system discovered two vulnerabilities revealed only when input values are well-formed.

In the current experiment, ACT was implemented on ActiveX controls; but it should also be applicable to other string processing programs whose source code is unavailable. Examples include network programs based on text-based protocols as well as programs which process text files.

This paper is organized as follows: Section 2 presents the algorithm used to auto-generate fuzzing grammars, as well as the basic principles involved. In Section 3, the implementation of the YMIR system, developed to carry out automatic fuzzing grammar generation via ACT, is described in detail. Section 4 demonstrates the usefulness of this system through enhanced code coverage and discovered vulnerabilities. An overview of related studies is given in Section 5. Section 6 discusses the limitations of the YMIR system, followed by the conclusion in Section 7.

2. Fuzzing grammar

ACT for a procedure produces a fuzzing grammar like following:

$\langle arg_x \rangle$ ::= "GET"⟨str⟩"HTTP/1.1"⟨str⟩

here, x is the index for parameters having a string data type, and ⟨str⟩ is a symbolic variable denoting that any string may be applied. In a fuzzing grammar, a symbolic variable is a path-independent field while a string constant affects an execution path. Therefore, a grammar-based fuzzer can enumerate concrete test cases by replacing symbolic variables with long strings, etc.

Table 1 – Four constraint generation rules.

#	Logs	Constraints	Similar functions
1	strcmp(str_k, "XY") ≠ 0	str_k ≠ "XY"	strcmpi, strcoll, lstrcmpA, strcmpiA, etc
2	strncmp(str_k, "XY", 2) ≠ 0	str_k ≠ "XY" str_l	_strnicmp, _strncoll, memcmp, etc
3	strstr(str_k, "XY") = 0	str_k ≠ str_l "XY" str_m	strchr, strrchr, etc
4	strtok_s(str_k, "XY",0) = str_k	str_k ≠ str_l "X" str_m AND str_k ≠ str_l "Y" str_m	strtok, strcspn, strpbrk, etc

2.1. Constraint generation rules

All SFCFs called in the process of executing the procedure under test are recorded in the log file. After the procedure under test is fully executed, the log file is analyzed to collect constraints on the execution path. If a symbolic constant (str_k) signifying that any string may be applied, is present as a parameter of the SFCF in the log file, it is identified as a constraint on the current path. As seen in Table 1, constraints are generated according to four rules. "XY" in Table 1 represents an arbitrary string constant.

If the constraint applied to current input is $c_1 \wedge c_2 \wedge \ldots \wedge c_n$ and ACT produces a new constraint c_{n+1}, the path constraint for a different execution path is $c_1 \wedge c_2 \wedge \ldots \wedge c_n \wedge \neg c_{n+1}$.

2.2. An example of a vulnerable procedure

Fig. 1 shows a sample pseudo code named the DoSomething procedure. The DoSomething procedure has a single string as input. The string preceding "*" designates an action to be taken, while the subsequent string is used as the parameter value needed to perform the action. When the DoSomething procedure is executed with "EXECUTE*http://update.web.server" as the input value, "http://update.web.server/update.exe" is downloaded and update.exe is run. Unfortunately, this procedure has a buffer overflow vulnerability caused by improper use of sprintf function in Line 6. Such error is triggered only by input strings beginning with "EXECUTE*http://" prefix whose length exceeds 1024 bytes. It is extremely unlikely that randomly generated test cases would satisfy such conditions and successfully reveal security vulnerability.

2.3. An example of fuzzing grammar generation

In this section, the DoSomething procedure in Fig. 1 is used to explain the process through which fuzzing grammars are generated by ACT.

2.3.1. First execution

For each string data type parameter in the procedure, ACT generates an initial fuzzing grammar signifying that any string may be applied. For the DoSomething procedure in Fig. 1, $\langle arg_1 \rangle ::= \langle str_1 \rangle$ is generated as the initial fuzzing grammar. When the DoSomething procedure is executed using str_1 (the concrete test case generated from the initial fuzzing grammar) as the first parameter value, the second and third lines in Fig. 1 are carried out. Since the symbolic constant (str_1) is used as the parameter for the SFCFs in Lines 2 and 3, constraints are generated according to Table 1.

Table 2 shows the new constraints and fuzzing grammars obtained during the first execution. The strtok_s function checks whether str_1 contains "*"; since it does not, the function returns the pointer for the entire string (str_1) entered. As a result, $str_1 \neq str_2$ "*" str_3 is collected as the constraint for the current path. To select a different path, this constraint is negated and is combined with the fuzzing grammar ($\langle arg_1 \rangle ::= \langle str_1 \rangle$) that generated the current test case (str_1) to yield a new fuzzing grammar ($\langle arg_1 \rangle ::= \langle str_2 \rangle$"*"$\langle str_3 \rangle$).

The strcmp in Line 3 checks whether str_1 is identical to "EXECUTE"; since it is not, it returns a value other than 0. Hence, $str_1 \neq$ "EXECUTE" is collected as the constraint. The negated constraint ($\langle str_1 \rangle ::=$ "EXECUTE") merges with the fuzzing grammar used to generate the current test case, thus yielding the new fuzzing grammar, $\langle arg_1 \rangle ::=$ "EXECUTE". Through this process, the first execution produces two fuzzing grammars.

```
01: void DoSomething(char *pszInputString) {
02:     pszCommand = strtok_s(pszInputString, "*", &pszNextToken);
03:     if(strcmp(pszCommand, "EXECUTE") == 0) {
04:         if(strncmp(pszNextToken, "http://", 7) == 0) {
05:             char szURL[1024];
06:             sprintf(szURL, "%s/update.exe", pszNextToken);
07:             DownloadAndExecute(szURL);
08:         }
09:     }
10: }
```

Fig. 1 – Pseudo code of the vulnerable procedure.

Table 2 – New fuzzing grammars generated at the first execution.

#	Logged API	New constraints	New fuzzing grammars
3	strtok_s(str_1, "*", 0) = str_1	$str_1 \neq str_2$ "*" str_3	$\langle arg_1 \rangle ::= \langle str_2 \rangle$ "$*$" $\langle str_3 \rangle$
4	strcmp(str_1, "EXECUTE") ≠ 0	$str_1 \neq$ "EXECUTE"	$\langle arg_1 \rangle ::=$ "EXECUTE"

The new fuzzing grammars generated during the first execution are used to produce test cases for executing new paths. Accordingly, two new test cases—i.e. str_2 "*" str_3 and "EXECUTE"—are generated, and these are used in turn to re-execute the DoSomething procedure.

2.3.2. *Second execution*

The second execution is carried out for the two test cases generated in the first execution, but there is only space to examine one of them—str_2 "*" str_3—here. As in the first execution, the second execution only carries out Lines 2 and 3 of the DoSomething procedure. Table 3 shows the new constraints and fuzzing grammars obtained at this execution. The first parameter (str_2 "*" str_3) and the return value (str_2) in the strtok_s function are different, indicating that the first parameter already contains the string "*". Hence, the strtok_s function is judged as a constraint that has already been reflected in the current test case, and is not collected as a new constraint. In the strcmp function, $\langle str_2 \rangle \neq$ "EXECUTE" is generated as the new constraint. The negated constraint is combined with the fuzzing grammar ($\langle arg_1 \rangle ::= \langle str_2 \rangle$"$*$"$\langle str_3 \rangle$) applied in the current test case, to produce $\langle arg_1 \rangle ::=$ "EXECUTE$*$"$\langle str_3 \rangle$ as the new fuzzing grammar.

Completing the second execution for test case str_2 "*" str_3 results in a single new fuzzing grammar. The test case generated from this new grammar, "EXECUTE*" str_3, is then used to re-execute the DoSomething procedure.

2.3.3. *Third execution*

When the third execution is carried out using "EXECUTE*" str_3, Lines 2, 3, and 4 from Fig. 1 are executed. Table 4 lists the parameters and return values for the three SFCFs thus executed. The strtok_s function is not collected as a new constraint for the same reason as we have explained in the second execution. The strcmp is not collected as a new constraint since it does not contain any symbolic constant (str_k), as a parameter. In the strncmp function, the condition that str_3 does not begin with "http://" is collected as a constraint. Therefore, this is combined with the fuzzing grammar ($\langle arg_1 \rangle ::=$ "EXECUTE$*$"$\langle str_3 \rangle$) used to generate the current test case to produce the new fuzzing grammar, $\langle arg_1 \rangle ::=$ "EXECUTE$*$http://"$\langle str_4 \rangle$.

2.3.4. *Fourth execution*

Carrying out the fourth execution for "EXECUTE*http://"str_4 results in the execution of Lines 2, 3, 4, 5, 6, and 7. This is a test case that can cover paths containing a buffer overflow vulnerability. Also, no more new constraints are collected, as seen in Table 5, meaning that ACT terminates at this point.

Fig. 2 lists all the fuzzing grammars generated through the four execution. In a fuzzing grammar, $\langle str \rangle$ is a symbolic variable that can become any string, while a constant string is a field that must not be altered in order for a specific path to be followed. Therefore, a grammar-based fuzzer can generate concrete test cases for revealing a security bug by replacing $\langle str \rangle$ with values intended to trigger vulnerabilities, such as long strings, etc. In this case, a concrete test case that substitutes $\langle str \rangle$ from the fifth fuzzing grammar ($\langle arg_1 \rangle ::=$ "EXECUTE$*$http://"$\langle str \rangle$) with a string exceeding 1024 bytes can trigger a buffer overflow vulnerability in the DoSomething procedure.

It is important to note that all fuzzing grammars, not just the most refined fifth fuzzing grammar, are meaningful. Although, in this particular example, the fifth fuzzing grammar is the one that produces the input value capable of triggering a vulnerability, each of the other fuzzing grammars also produces an input value for testing its own particular path.

2.4. *Algorithm for fuzzing grammar generation*

Algorithm 1. Algorithm for generating fuzzing grammars.

```
   param: P(procedure under test) , dt(data types of parameters in P)
   result: fg(fuzzing grammars)
01: procedure GenerateFuzzingGrammar(P, dt):
02: pc := true
03: for i := 1 to size(dt) do
04:       if dt[i] = string data type then pc := pc ∧ (⟨arg_i⟩ ::= ⟨str_i⟩)
05: fg := {pc}
06: testcaseList := GenConcreteTC(fg)
07: while testcaseList not empty do
08:       (input, currentPC) := GetFirstItem(testcaseList)
09:       newFGList := ACT(P, input, currentPC)
10:       fg := fg ∪ newFGList
11:       testcaseList := testcaseList ∪ GenConcreteTC(newFGList)
12: return fg

   parameters: P(procedure under test), input, pc(path constraints)
   result      : fg(fuzzing grammars)
13: procedure ACT(P, input, pc):
14: monitoredAPIList := ExecuteProcedureAndMonitorAPIs(P, input)
15: fg := ∅
16: for i := 1 to size(monitoredAPIList) do
17:       constraint := Rule-basedMapping(monitoredAPIList[i])
18:       fg := fg ∪ {pc ∧ ¬constraint}
19: return fg
```

Algorithm 1 illustrates an algorithm to generate fuzzing grammars. Each parameter of the procedure $\mathscr{P}$ under test is checked whether it has string data type. Constraints meaning that each string data type parameter may have an arbitrary string, are generated. An initial fuzzing grammar is produced by taking the conjunction of all generated constraints (from Line 2 through Line 5). In Line 6, concrete test cases are generated from the initial fuzzing grammars in order to carry

Table 3 – New fuzzing grammar generated at the second execution.

#	Logged API	New constraints	New fuzzing grammars
3	strtok_s(str_2 "*" str_3, "*", 0) = str_2	–	–
4	strcmp(str_2,"EXECUTE") ≠ 0	str_2 ≠ "EXECUTE"	$\langle arg_1 \rangle$ ::= "EXECUTE * "$\langle str_3 \rangle$

Table 4 – New fuzzing grammar generated at the third execution.

#	Logged API	New constraints	New fuzzing grammars
3	strtok_s("EXECUTE*"str_3, "*", 0) = "EXECUTE"	–	–
4	strcmp("EXECUTE", "EXECUTE") = 0	–	–
6	strncmp(str_3, "http://",7) ≠ 0	str_3 ≠ "http://"str_4	$\langle arg_1 \rangle$ ::= "EXECUTE * http : //"$\langle str_4 \rangle$

out ACT. This process involves substituting each symbolic variable in the fuzzing grammar with a symbolic constant. The symbolic constant is a predefined string constant to signify that any string may be applied. A string which does not seem to appear, is used as a symbolic constant.

Next, using the first item taken from the testcaseList as the input value (Line 8), ACT is performed for the procedure $\mathscr{P}$ (Line 9). ACT produces new fuzzing grammars (Line 10). Finally, newly-generated fuzzing grammars are used to produce new test cases, which are added to the testcaseList (Line 11). The steps from Line 7 through Line 11 are repeated until the testcaseList becomes empty.

During ACT, SFCF calls are monitored (Line 14). Constraints are collected from monitored SFCF calls using Table 1 (Line 17). To generate new fuzzing grammars, collected constraints are negated one by one and the conjunction is taken of each negated constraint with the path constraint applied to the current input value (Line 18). Test cases generated from new fuzzing grammars will execute different paths of the procedure $\mathscr{P}$.

3. Implementation of the YMIR system

We created the YMIR[1] System, which automatically generates fuzzing grammars for ActiveX controls. ActiveX controls are highly useful in demonstrating the effectiveness of the proposed approach because 1) they are real-world binary programs whose features include string type data processing; and 2) the majority (e.g., 22 out of 24) of their vulnerabilities, disclosed by exploit-db.com between January and October 2012 (Exploit database), occurred while processing string inputs.

As illustrated in Fig. 3, the YMIR system takes an ActiveX control as the input and produces fuzzing grammars for testing the ActiveX control.

The fuzzing grammars produced by the YMIR system can be used by a grammar-based fuzzer to reveal security bugs. The YMIR system does not require the source code for the ActiveX control being tested, nor does it carry out time-consuming instruction logging and constraint solving.

The YMIR system begins by identifying C run-time library functions statically linked to the ActiveX control being tested. In the current study, a IDA Pro plugin was developed specifically for this purpose. IDA Pro uses Fast Library Identification and Recognition Technology (FLIRT) to analyze binary files and identify well-known library functions; FLIRT technology works by pre-generating signatures for the library functions of well-known compilers, and carrying out pattern matching to identify the functions in the target binary file (Guilfanov). The YMIR system executes IDA Pro to analyze the ActiveX control under test; when this is completed, the IDA Pro plugin extracts the location of the files containing SFCFs from among the library functions identified by IDA Pro. At present, the IDA Pro plugin developed in this study is capable of identifying 78 types of SFCFs included in the C run-time library.

Next, the YMIR system analyzes the type library of the target ActiveX control and extracts the procedure information to be tested. Using ACT, the system performs fuzzing grammar generation for procedures containing string data type parameters.

4. Evaluation

In this section, we demonstrate two experiment results. First, our approach achieves higher code coverage than random fuzz testing. Second, our technique is able to disclose new vulnerabilities triggered only by well-formed input values.

4.1. *Enhanced code coverage*

The branch coverage metric was used for measuring code coverage. As it is difficult to accurately identify all branches in an ×86 binary program, the relative difference between the branches covered by random fuzz testing and those covered by the YMIR system was compared. To count the branches covered by each technique, a IDA Pro debugger plugin was developed. This plugin logged all instructions executed by each testing technique, as well as the relative addresses of the instructions. This is a time-consuming process, since it requires instruction tracing, as is the case with instruction-level

[1] Ymir is the name of a character in Norse mythology. Legend tells that his body was used to create Midgard: his flesh became the earth, his blood formed seas and lakes, and his bones became the mountains. We named our new tool the "YMIR" system because the process in which Ymir's body was divided and re-materialized is analogous to the refinement process for fuzzing grammars.

Table 5 – No fuzzing grammar generated at the fourth generation.

#	Logged API	New constraints	New fuzzing grammars
3	strtok_s("EXECUTE*http://"str_4, "*", 0) = "EXECUTE"	–	–
4	strcmp("EXECUTE", "EXECUTE") = 0	–	–
6	strncmp("http://"str_4, "http://", 7) = 0	–	–

concolic testing. However, in this case, instruction tracing was performed for the purpose of evaluation only; in actual fuzzing grammar generation, it is dispensed with altogether.

The process through which branch coverage was measured from the logged instructions is as follows: First, conditional branch statements such as jz and jb were searched for among the logged instructions. The relative address containing a conditional branch statement and that of the immediately following instruction were bound into a pair and counted as a single branch. A branch is counted as covered only when the conditional branch statements are contained within the DLL file of the target ActiveX control. That is, branches occurring in kernel32.dll or the library DLL, which are basic modules provided by the OS, are not counted.

As shown in Fig. 4, if the execution of the jnz instruction in the relative address 1308h is followed by the execution of the mov instruction in 130Ah (1308h, 130Ah), becomes a pair and the branch is counted as covered. Two branches—(1308h, 130Ah) and (1308h, 1333h)—exist in the example shown in Fig. 4.

The evaluation technique described thus far was used to perform experiments on various ActiveX control procedures. Strings of the same number and length as the test cases produced by fuzzing grammars were randomly generated and used as test cases for random fuzz testing.

The c procedure of YTHelper2[2] takes a single string data type as its parameter. Running the YMIR system for this procedure resulted in the generation of 16 fuzzing grammars, as listed in Fig. 5.

Fig. 6 shows the difference between the branch coverage of YMIR-enabled testing and random fuzz testing. As can be seen here, YMIR-enabled testing covered a total of 137 branches, whereas random fuzz testing covered only 91. In short, the former technique was able to cover 50% more branches than the latter.

A comparison of the number of branches covered by each test case clearly demonstrates the superiority of YMIR-enabled testing. The initial test case generated by the YMIR system was str_1. In this case, fuzzing grammar-based testing covered 84 branches, while random fuzz testing covered 83. Among these, 82 branches were identical. That a small number of non-overlapping branches were covered—two by the YMIR system and one by random fuzz testing—appears to be the result of execution on non-identical strings.

The second test case generated by the YMIR system was str_2"|"str_3. At this point, the number of branches covered by YMIR-enabled testing increased by six, for a total of 90. What is interesting here is that the same increase in the number of branches is observed for random fuzz testing as well, with 89 branches covered during the second test case. This was caused by the coincidental inclusion of "|" in the randomly-generated second test case. Thereafter, the 13th test case for random fuzz testing can be seen to increase coverage by two branches, but overall coverage remains restricted to virtually the same branches. In stark contrast, YMIR-enabled testing covers 40 new branches with the third test case ("gv"), four new branches with the fourth test case ("wr"), and three new branches with the fifth test case ("onw"), at which point coverage increase can no longer be observed.

These results demonstrate that random fuzz testing can easily cover branches that are reachable through the presence of a single character, but not when two or more characters need to be present consecutively. The probability that one of 16 randomly-generated test cases for the c procedure of YTHelper2 will begin with the string "gv" is only 0.02% (16*(1/256)*(1/256)*100). The longer the string requiring matching is, the slimmer the possibility becomes that random fuzz testing will cover the branch in question.

Table 6 summarizes the differences in branch coverage between random fuzz testing and YMIR-enabled testing for a variety of ActiveX controls. As the table shows, YMIR-enabled testing has a 15–50% higher rate of branch coverage than random fuzz testing.

4.2. *Discovered vulnerabilities*

We implemented the simple fuzzer that uses fuzzing grammars. The fuzzer creates concrete test cases by replacing each path-independent symbolic variable($\langle str \rangle$) of fuzzing grammars with a long string and executes ActiveX control procedures with them. Since the fuzzer contains a built-in debugger, it detects any exception that may occur while procedures are running. Using the fuzzing grammars generated by YMIR, we were able to discover two significant vulnerabilities.

Our fuzzer discovered an uninitialized heap reference vulnerability triggered when the first argument of Fath VideoEditX's DetectClips procedure (Fath Software) contains ".wmv," ".wma," or ".asf."

$$
\begin{aligned}
&1:\ \langle arg_1 \rangle ::= \langle str \rangle \\
&2:\ \langle arg_1 \rangle ::= \langle str \rangle \text{ “*” } \langle str \rangle \\
&3:\ \langle arg_1 \rangle ::= \text{“EXECUTE”} \\
&4:\ \langle arg_1 \rangle ::= \text{“EXECUTE*”} \ \langle str \rangle \\
&5:\ \langle arg_1 \rangle ::= \text{“EXECUTE*http://”} \ \langle str \rangle
\end{aligned}
$$

Fig. 2 – All fuzzing grammars generated for the DoSomething procedure.

[2] YTHelper2 is an ActiveX control that is installed along with Yahoo! Messenger (ver. 10.0.0.1102).

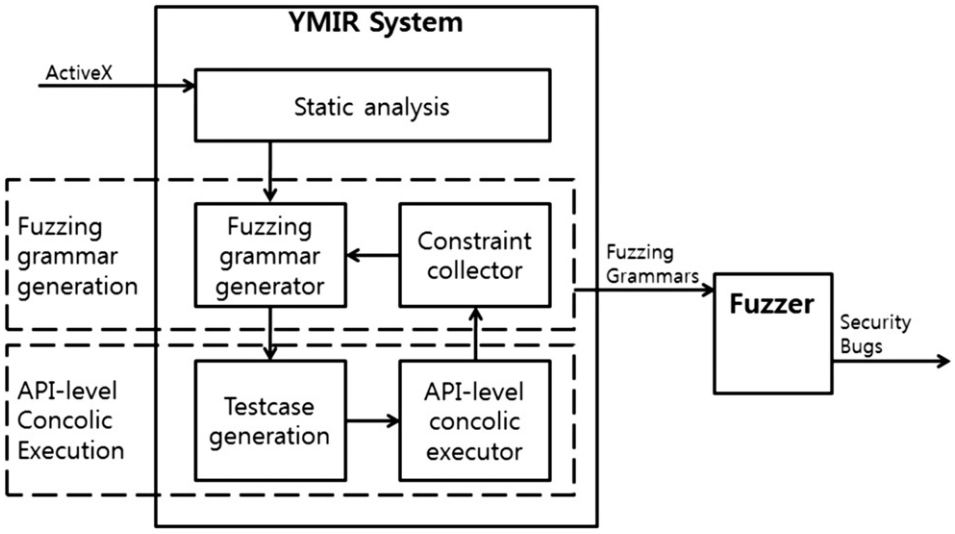

Fig. 3 – Architecture of the YMIR system.

In addition, our fuzzer discovered a vulnerability that can generate an arbitrary registry key in the SetInstallInformation2 procedure of DestinyInstallX (Cyberdigm). The results of fuzzing grammar generation for the SetInstallInformation2 procedure are given in Fig. 7. By running the SetInstallInformation2 procedure for the ninth fuzzing grammar, the registry key with the name of "str_2" can be created under the specific registry key.

5. Related work

5.1. *Blackbox fuzz testing*

It is difficult for blackbox fuzz testing to reveal vulnerabilities triggered by a well-formed input value since blackbox fuzz testing uses preset or randomly generated test cases to test a target program. However, blackbox fuzz testing is still prevalent in real-world because it is fast and can be implemented easily.

All existing fuzzers for ActiveX controls use blackbox fuzz testing, which involves substituting input values frequently used to trigger vulnerabilities, such as lengthy strings, file paths, urls, etc into string data type parameters (Dormann and Plakosh; Moore; iDefense Labs; Bret-Mounet; shaneh).

Our approach can overcome low code coverage of blackbox fuzz testing, and disclose vulnerabilities triggered only by a well-formed input value.

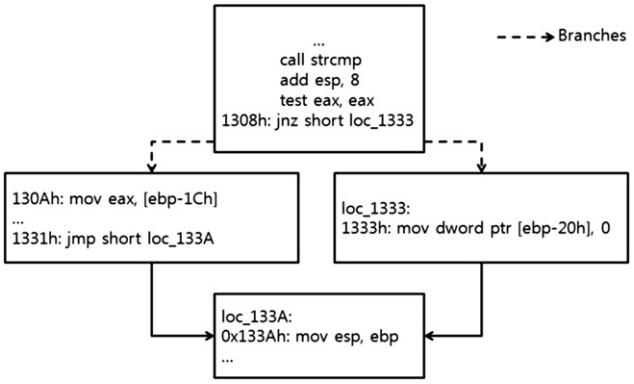

Fig. 4 – Counting of covered branches.

5.2. *Grammar-based fuzz testing*

Peach (Eddington) and Sulley (Amini and Portnoy) proposed a fuzzing framework in which analysts specify input format such as data types of and dependencies among fields. Analysts use specification or perform reverse engineering process to manually derive information necessary to specify data representation. Fuzz testing conducted using test cases generated based on such information would achieve higher coverage than randomly generated test cases would and is likely to detect more vulnerabilities in real-world applications (Neystadt, 2008).

Our proposed technique automates such labor-intensive process.

5.3. *Concolic testing*

Concolic testing (Sen et al., 2005) is also known as dynamic symbolic execution (Cadar et al., 2008a, 2008b; Godefroid et al., February 2008; Molnar and Wagner, 2007). Concolic testing generates concrete test cases to cover various paths by interleaving concrete execution with symbolic execution. Given a program $\mathscr{P}$, concolic testing runs $\mathscr{P}$ with a random concrete input value. After $\mathscr{P}$ terminates, all executed instructions are analyzed to collect constraints. When part of the input value is used as a branch condition, the branch condition is collected as a constraint. In Fig. 8, the current path constraint is expressed as $c_1 \wedge c_2 \wedge c_3$ when branch constraints c_1, c_2, and c_3 are executed in order. These constraints are one by one negated, and constraint solving is carried out to generate new concrete test cases for exploring new paths. In

$\langle arg_1 \rangle ::= \langle str_1 \rangle$
$\langle arg_1 \rangle ::= \langle str_2 \rangle$ "|" $\langle str_3 \rangle$
$\langle arg_1 \rangle ::=$ "gv"
$\langle arg_1 \rangle ::=$ "wr"
$\langle arg_1 \rangle ::=$ "onw"
$\langle arg_1 \rangle ::= \langle str_2 \rangle$ "|" $\langle str_4 \rangle$ "|" $\langle str_5 \rangle$
$\langle arg_1 \rangle ::=$ "gv|" $\langle str_3 \rangle$
$\langle arg_1 \rangle ::=$ "wr|" $\langle str_3 \rangle$
$\langle arg_1 \rangle ::=$ "onw|" $\langle str_3 \rangle$
$\langle arg_1 \rangle ::= \langle str_2 \rangle$ "|" $\langle str_4 \rangle$ "|" $\langle str_6 \rangle$ "|" $\langle str_7 \rangle$
$\langle arg_1 \rangle ::=$ "gv|" $\langle str_4 \rangle$ "|" $\langle str_5 \rangle$
$\langle arg_1 \rangle ::=$ "wr|" $\langle str_4 \rangle$ "|" $\langle str_5 \rangle$
$\langle arg_1 \rangle ::=$ "onw|" $\langle str_4 \rangle$ "|" $\langle str_5 \rangle$
$\langle arg_1 \rangle ::=$ "gv|" $\langle str_4 \rangle$ "|" $\langle str_6 \rangle$ "|" $\langle str_7 \rangle$
$\langle arg_1 \rangle ::=$ "wr|" $\langle str_4 \rangle$ "|" $\langle str_6 \rangle$ "|" $\langle str_7 \rangle$
$\langle arg_1 \rangle ::=$ "onw|" $\langle str_4 \rangle$ "|" $\langle str_6 \rangle$ "|" $\langle str_7 \rangle$

Fig. 5 – Results of fuzzing grammar generation for YTHelper2's c procedure.

Fig. 8, three constraint solving is performed for $c_1 \wedge c_2 \wedge \neg c_3$, $c_1 \wedge \neg c_2$, and $\neg c_1$. Again, concolic testing runs $\mathscr{P}$ with new test cases. This process continues until all paths of $\mathscr{P}$ are covered or given time expires.

Although concolic testing can greatly enhance code coverage, instruction logging and constraint solving are still time-consuming (Godefroid et al., February 2008).

5.4. *Symbolic grammars*

CESE (Concolic Execution with Selective Enumeration) transfers as a symbolic grammar the grammar that a given input value can have, and uses an enumerative technique to enumerate string sets that are valid for the symbolic grammar (Majumdar and Xu, 2007). In this way, the respective disadvantages of symbolic execution (requiring a lengthy investment of time) and the enumerative technique (tending to generate a large number of redundant test cases for the same path) can be overcome. CESE also assumes grammars themselves to be pre-given, unlike the technique proposed in our study.

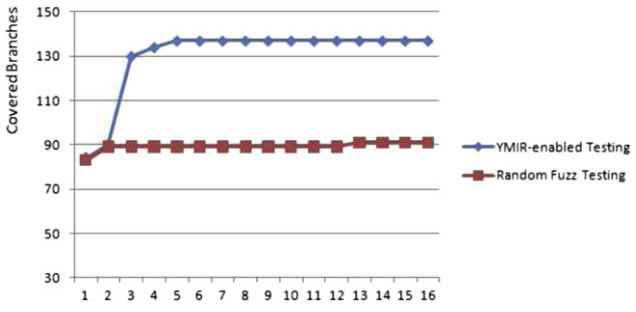

Fig. 6 – Comparison of branch coverage between YMIR-enabled testing and random fuzz testing.

5.5. *Other studies*

Autodafé (Vuagnoux) uses a technique called "weighting attacks with markers." This technique works by inserting markers in strings or values that can be controlled by the user, and using a debugger during execution to check if these markers are conveyed to functions, like strcpy, that are known to be unsafe. If a marker ends up being conveyed to such a function, a lengthy string can be entered in place of the marker to trigger vulnerabilities, including buffer overflow. If a partial description of the format that can be adopted by a protocol or input string is provided, Autodafé uses each canonical element as a marker. The technique proposed in our study differs fundamentally from this in that it automatically generates fuzzing grammars, which correspond to such partial descriptions.

Flayer is basically a tool designed to aid manual code auditing (Drewry and Ormandy, 2007). Rather than attempting to maximize code coverage, the auditor conveniently eliminates the constraints in the target program and assists testing. However, if a path is found to contain a vulnerability beyond the constraints of the program, the task of generating a test case to reach the path in question is left entirely to the auditor. What distinguishes the technique proposed in the current study is that it generates fuzzing grammars capable of automatically producing test cases for going beyond the constraints of a given program.

BuzzFuzz (Ganesh et al., 2009) performs directed whitebox fuzz testing with the source code of the target program, a list of potentially vulnerable locations, and seed inputs. BuzzFuzz traces taint information during execution of the instrumented target program. If some input bytes are used in any potentially vulnerable location, they are modified to generate new inputs. BuzzFuzz re-executes the target program with new inputs which is expected to reveal a vulnerability. This approach is

Table 6 – Comparison of branch coverage by random fuzz testing and YMIR-enabled testing.

ActiveX name	DjVuCtl (DjVu browser plug-in)	AxisMedia control	Chilkat OmaDrm (Chilkat Encryption ActiveX)	SqliteDb (EztoolsLib2)
Procedure name	PrintDocTo	Set additional MediaSource	SetEncoded IV	Init
# of fuzzing grammars	96	8	10	22
# of branches covered by r.f.t	75	242	314	361
# of branches covered by YMIR	87	286	472	416
Differences	+12(16%)	+44(18%)	+158(50%)	+55(15%)

$$
\begin{aligned}
&01:\ \langle arg_2\rangle ::= \langle str\rangle, \langle arg_3\rangle ::= \langle str\rangle\\
&02:\ \langle arg_2\rangle ::= \langle str\rangle\ \text{“\#version”}\ \langle str\rangle, \langle arg_3\rangle ::= \langle str\rangle\\
&03:\ \langle arg_2\rangle ::= \langle str\rangle\ \text{“=”}\ \langle str\rangle, \langle arg_3\rangle ::= \langle str\rangle\\
&04:\ \langle arg_2\rangle ::= \langle str\rangle, \langle arg_3\rangle ::= \langle str\rangle\ \text{“:”}\ \langle str\rangle\\
&05:\ \langle arg_2\rangle ::= \langle str\rangle\ \text{“\#version”}\ \langle str\rangle\ \text{“=”}\ \langle str\rangle, \langle arg_3\rangle ::= \langle str\rangle\\
&06:\ \langle arg_2\rangle ::= \langle str\rangle\ \text{“=”}\ \langle str\rangle\ \text{“\#version”}\ \langle str\rangle, \langle arg_3\rangle ::= \langle str\rangle\\
&07:\ \langle arg_2\rangle ::= \langle str\rangle\ \text{“\#version”}\ \langle str\rangle, \langle arg_3\rangle ::= \langle str\rangle\ \text{“:”}\ \langle str\rangle\\
&08:\ \langle arg_2\rangle ::= \langle str\rangle\ \text{“=”}\ \langle str\rangle, \langle arg_3\rangle ::= \langle str\rangle\ \text{“:”}\ \langle str\rangle\\
&09:\ \langle arg_2\rangle ::= \langle str\rangle\ \text{“\#version”}\ \langle str\rangle\ \text{“=”}\ \langle str_2\rangle, \langle arg_3\rangle ::= \langle str\rangle\ \text{“:”}\ \langle str\rangle\\
&10:\ \langle arg_2\rangle ::= \langle str\rangle\ \text{“=”}\ \langle str\rangle\ \text{“\#version”}\ \langle str\rangle, \langle arg_3\rangle ::= \langle str\rangle\ \text{“:”}\ \langle str\rangle
\end{aligned}
$$

Fig. 7 – Fuzzing grammars generated for DestinyInstallX's SetInstallInformation2 procedure.

different from our technique because it needs the source code of the target program and inputs are not generated for the purpose of increasing the code coverage.

At present, whitebox fuzzing, like SAGE, faces clear limitations when used to test applications with highly-structured inputs, such as compilers and interpreters. The main problem is that too many paths exist during the early processing stage, making it impossible to move on to the subsequent stage. To overcome this problem, whitebox fuzzing can be performed using grammar-based specifications for valid input values (Godefroid et al., June 2008). Grammar-based whitebox fuzzing marks tokens returned by tokenization functions as symbolic variables, extracts as constraints the effects such tokens have on program paths, and generates new input values by utilizing a context-free constraint solver to solve the extracted constraints. Unfortunately, such a process cannot be developed for general use, because it is dependent on individual programs. It also diverges fundamentally from our proposed technique, which is designed to generate fuzzing grammars, in that grammars are assumed to have been already provided.

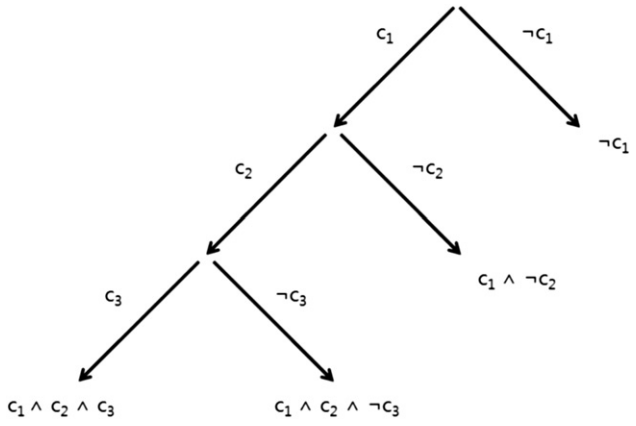

Fig. 8 – Example of path constraints.

6. Limitation

The technique proposed in this paper does not offer any efficacy for programs using simply-formatted strings as input. In such cases, sufficient testing can be performed using the existing blackbox fuzzing technique. Also, when the developer wishes to check the format of input strings using self-implemented code rather than well-known library functions, the proposed technique cannot generate fuzzing grammars that enhance code coverage. However, in many instances developers tend to prefer using well-known, proven library functions.

For the technique proposed in this study, the accuracy of the fuzzing grammars generated increases in proportion to the number of SFCFs monitored. However, there are various types of SFCFs. A broad variety of libraries, such as the Standard Template Library (STL) and the MFC Library, etc, currently exist, not to mention that some DLLs provided by operating systems also contain exported SFCFs. Therefore, all

of these should be identified and their semantics should be analyzed. Fortunately, these functions are implemented in ways that are very similar to the functions already monitored by the YMIR system. Thus, further extension is only a matter of time; no technical difficulties are involved. In addition, many developers tend to show a preference for a limited number of library functions; excluding a number of lesser-known library functions does not greatly impact the accuracy of fuzzing grammars.

7. Conclusion

Recently, grammars for input values are being frequently adopted as a means of overcoming the limits of blackbox fuzz testing. However, existing studies are substantially impeded by the feature that grammar generation is carried out manually. To overcome the problems of existing studies, the current study proposes automatic fuzzing grammar generation through API-level concolic testing. Because fuzzing grammars explicitly differentiate fields that affect paths from those that do not, they can be used to generate concrete test cases that can easily trigger security bugs while covering diverse paths, by substituting path-independent fields with long strings, etc.

We developed the YMIR system to demonstrate feasibility of the proposed concept. To the best of our knowledge, the YMIR system is the first tool ever developed to carry out whitebox-based fuzzing of ActiveX controls. The YMIR system is capable of automatically generating fuzzing grammars for ActiveX control procedures that process string data type input values.

The experiment results showed that fuzzing grammars increased code coverage in programs using highly-structured strings as input by approximately 15–50% in comparison to randomly generated test cases. In addition, the YMIR system discovered two significant vulnerabilities revealed only when input values are well-formed.

At present, the YMIR system is designed to generate fuzzing grammars that can be used by ActiveX control fuzzers. However, it can also be applied to other programs that processes input strings even if source code is unavailable.

Acknowledgements

This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST)/National Research Foundation of Korea (NRF) (Grant 2012-0000473). This work was supported by the DGIST MIREBrian (Operation of CPS Global Center) of the Ministry of Education, Science and Technology of Korea.

REFERENCES

Aitel D. The advantages of block-based protocol analysis for security testing, http://www.immunitysec.com/resources-papers.shtml; 2002.

Amini P, Portnoy A. Sulley – fuzzing platform. http://code.google.com/p/sulley/.

AXIS media control. http://www.axis.com/techsup/cam_servers/dev/activex.htm.

Barrett C, Berezin S. CVC Lite: a new implementation of the cooperating validity checker. In: Proceedings of the 16th international conference on computer aided verification; July 2004.

Bret-Mounet F. COMBust. http://www.blackhat.com/presentations/bh-usa-03/bh-us-03-bretmounet-combust.zip.

Cadar C, Ganesh V, Pawlowski P, Dill D, Engler D. EXE: automatically generating inputs of death. ACM Transactions on Information and System Security 2008a;12(2).

Cadar C, Dunbar D, Engler D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: USENIX symposium on operating systems design and implementation; 2008b.

Chilkat Encryption ActiveX. http://www.chilkatsoft.com/crypt-activex.asp.

Cyberdigm. DestinyInstallX. http://cyberdigm.co.kr/english/index.html.

DjVu browser plug-in. http://www.djvu.org/.

Dormann W, Plakosh D. Dranzer. http://www.cert.org/vuls/discovery/dranzer.html.

Drewry W, Ormandy T. Flayer: exposing application internals. First workshop on offensive technologies; August 2007.

Eddington M. Peach fuzzing platform. http://peachfuzzer.com/.

Exploit database. http://www.exploit-db.com/.

EztoolsLib2. http://www.eztools-software.com/home/downloads.asp.

Fath Software. VideoEditX. http://www.fathsoft.com/VideoEditX.html.

Ganesh V, Leek T, Rinard M. Taint-based directed whitebox fuzzing. In: Proceedings of the 31st international conference on software engineering; May 2009.

Godefroid P, Levin M, Molnar D. SAGE: whitebox fuzzing for security testing. ACM Queue January 2012;10(1).

Godefroid P, Levin M, Molnar D. Automated whitebox fuzz testing. In: Network and distributed system security symposium; February 2008.

Godefroid P, Kieun A, Levin M. Grammar-based whitebox fuzzing. In: Programming language design and implementation; June 2008.

Gorbunov S, Rosenbloom A. AutoFuzz: automated network protocol fuzzing framework. International Journal of Computer Science and Network Security August 2010;10(8).

Guilfanov I. Fast library identification and recognition technology. http://www.hex-rays.com/idapro/flirt.htm.

IDA Pro. http://www.hex-rays.com/products.shtml.

iDefense Labs. COMRaider. http://labs.idefense.com/software/fuzzing.php#more_comraider.

Kieun A, Ganesh V, Guo PJ. HAMPI: a solver for string constraints. In: ACM international symposium on testing and analysis; July, 2009.

Majumdar R, Xu R-G. Directed test generation using symbolic grammars. In: IEEE/ACM international conference on automated software engineering; November, 2007.

Molnar DA, Wagner D. Catchconv: symbolic execution and run-time type inference for integer conversion errors. Technical report UCB/EECS-2007-23. Berkeley: EECS Department, University of California; February 2007.

Moore HD. AxMan. http://digitaloffense.net/tools/axman/.

Neystadt J. Automated penetration testing with white-box fuzzing, http://msdn.microsoft.com/en-us/library/cc162782.aspx; February 2008.

Protos. http://www.ee.oulu.fi/research/ouspg/protos/.

Sen K, Marinov D, Agha G. CUTE: a concolic unit testing engine for C. In: Joint meeting of the European software engineering conference and ACM SIGSOFT symposium on the Foundations of Software Engineering; September 2005.

shaneh. axfuzz. http://sourceforge.net/projects/axfuzz/.

Tillmann N, de Halleux J. Pex – white box test generation for.NET. In: International conference on tests and proofs; April 2008.
Vuagnoux M. Autodafé: an act of software torture. http://autodafe.sourceforge.net/.
Yahoo! Messenger 10. http://messenger.yahoo.com/download/.
Ymir. http://en.wikipedia.org/wiki/Ymir.

Su Yong Kim is a senior member of the engineering staff in the attached institute of ETRI. His research focuses on finding vulnerabilities in softwares. He received his Ph.D. in Computer Science Department, KAIST in 2011.

Sungdeok (Steve) Cha is a professor in Computer Science and Engineering Department, Korea University. His research interests include software safety and computer security. Cha has a Ph.D. in information and computer science from the University of California, Irvine.

Doo-Hwan Bae is a professor in Computer Science Department, College of Information Science and Technology, KAIST. His research interests include process/quality improvement and software design methods. He received a Ph.D degree from University of Florida, in 1992.