Fuzzing the ActionScript Virtual Machine

Guanxing Wen, Yuqing Zhang, Qixu Liu, Dingning Yang
National Computer Network Intrusion Protection Center
School of Computer and Control, UCAS, Beijing 100049 PR China
wengx522@sina.com, {zhangyq, liuqixu}@ucas.ac.cn, yangdn@nipc.org.cn

ABSTRACT

Fuzz testing is an automated testing technique where random data is used as an input to software systems in order to reveal security bugs/vulnerabilities. Fuzzed inputs must be binaries embedded with compiled bytecodes when testing against ActionScript virtual machines (AVMs). The current fuzzing method for JavaScript-like virtual machines is very limited when applied to compiler-involved AVMs. The complete source code should be both grammatically and semantically valid to allow execution by first passing through the compiler. In this paper, we present *ScriptGene*, an algorithmic approach to overcome the additional complexity of generating valid ActionScript programs. First, nearly-valid code snippets are randomly generated, with some controls on instruction flow. Second, we present a novel mutation method where the former code snippets are lexically analyzed and mutated with runtime information of the AVM, which helps us to build context for undefined behaviours against compiler-check and produce a high code coverage. Accordingly, we have implemented and evaluated *ScriptGene* on three different versions of Adobe AVMs. Results demonstrate that *ScriptGene* not only covers almost all the blocks of the official test suite (Tamarin), but also is capable of nearly twice the code coverage. The discovery of six bugs missed by the official test suite demonstrates the effectiveness, validity and novelty of *ScriptGene*.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Testing tools

Keywords

Fuzz testing, Vulnerability discovery, ActionScript

1. INTRODUCTION

The theory of application virtual machines is well developed. An application virtual machine (VM) isolates the host OS from a single process. The VM is created when the associated process is started and destroyed when the associated process exits [27]. VM Implementations can be divided into two categories based on the code execution. JavaScript-like VMs accept source code directly, interpreting and executing code lines sequentially. ActionScript-like VMs accept only bytecodes generated by a compiler from source code. This additional compilation stage adds complexity to ActionScript-like VMs when compared to JavaScript-like VMs. Since many VMs are programmed using C/C++, it is not surprising that they contain bugs similar to other complex software systems.

Fuzz testing or fuzzing is a form of testing heavily used for finding security vulnerabilities in software. Since completely random strings generated by traditional fuzzing methods do not usually execute in VMs, grammar-based fuzzing is employed instead. Grammar-based whitebox approach of [17] has been used on a JavaScript VM. They presented a dynamic test-case generation algorithm, where symbolic execution directly generates grammar-based constraints, whose satisfiability is verified using a custom grammar-based constraint solver. The solver solves the altered constraints and directs code snippet generation to cover more execution paths of the VM. Unfortunately this approach is limited to JavaScript-like VMs, since feedback operates on the input directly. In comparison, ActionScript-like VMs only accept binaries embedded with bytecodes, which are the outputs of the compiler. Therefore, a hypothetical solver is unable to analyze and generate new pieces of code as inputs, rendering this approach ineffective. Grammar-based blackbox approach of [19] is yet another effective fuzzing method on VMs. Previously, this approach has also been restricted to testing JavaScript-like VMs. Both grammar-based whitebox and blackbox fuzzing methods focus only on the grammar, ignoring the substantial contributions of runtime class, property and API functions to the discovery of bugs in VMs.

To overcome the difficulties associated with fuzzing of ActionScript-like VMs, we have created *ScriptGene*. *ScriptGene* generates ActionScript (AS) source code, compiles and executes them on Adobe Flash ActionScript Virtual Machines (AVMs). As seen in figure 1, the left-hand side sequence of block diagrams, connected by downward arrows illustrates the *ScriptGene* code generation process. Prior to the code generation, a knowledge base is built by interpreting the grammar file of AS and the official test cases. Then, grammar-based blackbox methods generate nearly-valid code. The right-hand side sequence of the figure with upward arrows illustrates the mutation phase. A

ASIA CCS'13, May 8–10, 2013, Hangzhou, China.

lexier marks all the identifiers in source code, pointing to functions, classes and variables with markers, followed by mutations and substitutions of markers with actual runtime class information. Lastly, the test case programs are created and sent to the target AVM.

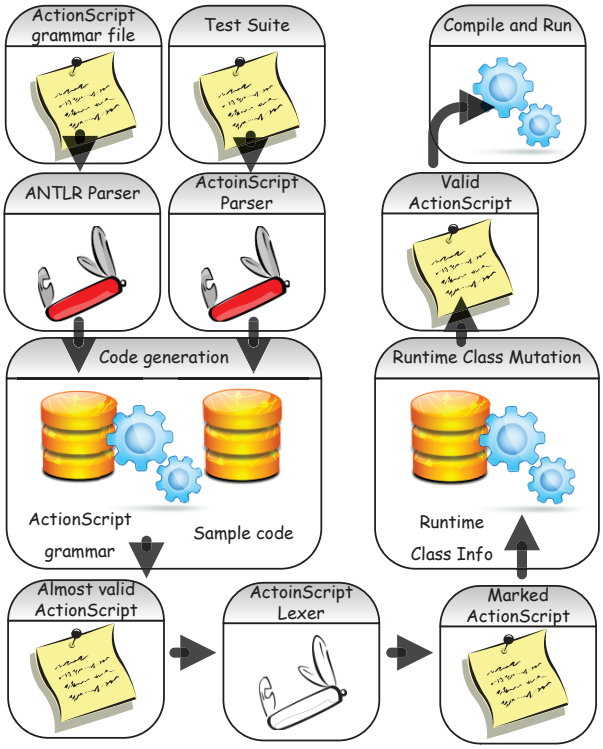

Figure 1: *ScriptGene* workflow.

In comparison with earlier approaches, *ScriptGene*'s test-generation techniques substantially advance the state of the art in fuzzing VMs by generating compilable and expressive random source code of high complexity, employing many AS language features. Compiler-validity is ensured by generating AS code that does not contain undefined classes, variables or functions; nor depend on random arrangements of the code structure. Currently, *ScriptGene* generates AS programs suitable for testing AVMs. Flexibility of our ideas underpinning *ScriptGene*, as well as our specific implementation should easily port to test other VMs. This paper summarizes the construction of *ScriptGene* and some of its current testing results.

Adobe Flash is a multimedia platform used to add animations to web pages and hybrid document formats such as PDFs and Office documents. Usually, Adobe Flash files with the ".SWF" (SWF) extension are treated as harmless files or as videos by the host OS. AS code inside SWFs are typically used to control animations and merely contain functions to operate natively on local resources (e.g., disk, process...). Thus, AVMs are running in trusted environments. If a SWF file triggers a vulnerability of the AVM and executes native code out of the trusted environment, the local resources could be compromised. Furthermore, Just-In-Time compilers (JIT) of AVMs can be used to breakthrough memory protection methods such as ASLR and DEP [16]. APT attack to RSA [15] in 2011 demonstrated the ramifications of AVM security vulnerabilities.

AVMs were chosen as the testing targets for two reasons. First and foremost, vulnerabilities in trusted environments are intrinsically harmful, more so than an equivalent language that operates on the local resources natively. Second, there appears to be a lack of existing grammar-based approaches focusing on ActionScript-like languages, where compilers are involved.

- Our first contribution is the implementation of a grammar-based blackbox method to fuzz test complier-involved AVMs. Our implementation has covered almost all the code blocks (>96%) of the official test cases.
- Our second contribution is the advancement of the state of the art in grammar-based blackbox test code generation, coupled with runtime class mutations. Our approach not only has allowed us to gain additional code coverage to nearly double that of the official test cases in our experiments, but also has discovered six previously undocumented bugs in three versions of AVMs [11] that were missed by the official test cases.

The remainder of this paper is organized as follows. Section 2 presents the technical details on *ScriptGene*, from the code generation phase to the runtime class mutation phase. Section 3 details the actual implementation and discusses our evaluations. Section 4 discusses related works and the state of the art in fuzz testing. Section 5 details the simplified strategy for *ScriptGene* and some of its limitations. Section 6 concludes the paper.

2. SCRIPTGENE

The basic concepts behind *ScriptGene* are in Section 2.1. The additional grammar controls are in Section 2.2. Our AS code generation algorithm is presented in Section 2.3. Extraction of runtime information for runtime class mutations is detailed in Section 2.4. Mutation of extracted information into code to create compilable source code is presented in Section 2.5.

2.1 Basic concepts and terminology

ActionScript and AVMs There are currently two versions of AS: AS2 and AS3, executed by AVM1 and AVM2. AS2 lacks many new features, such as 3D presentations and is less effective in code execution as compared to AS3. AS3 is chosen, since it is the version most frequently employed by current SWFs. AS3 based on ECMAScript4 standard [6] is an object-oriented programming language used to build SWFs. Unless otherwise noted, AS refers to AS3 and AVM refers to AVM2 hereafter.

Grammar files In order to generate and parse source code, the grammar file [7] written in ANTLR, containing the grammar standards of AS is utilized. ANTLR is a parser generator framework written by Parr and Quong [23]. Once the grammar file is established, the task of building a parser is similar to compiling source code. In addition to the AS parser, an ANTLR language parser parses the ANTLR format AS grammar file, since we wish to generate AS code snippets beyond merely parsing them. Since some features of ANTLR are only enabled under Java, Java is used to parse source code. Other parts of *ScriptGene* are programmed in Python, to leverage its string manipulation capabilities.

In total, there are three grammar files for *ScriptGene*: a grammar file used to build a parser for AS (G1), a grammar file used to guide the AS code generation process (G2) and a grammar file used to build a parser for ANTLR (G3). G2 is a modified version of G1, which will be explained in Section 2.2. G3 is the official ANTLR grammar file inside the example directory of its Java distributions.

Grammar rules in ANTLR format In ANTLR, AS grammar elements can be divided into three categories: rules, blocks and atoms. Rules comprise the main structures listed in an ANTLR format grammar file. Rules also contain sub-rules. Sub-rules are mixtures of blocks and atoms. Blocks consist of atoms. Atoms represent terminal and non-terminal nodes. Non-terminal nodes are other rules and terminal nodes are tokens or real characters inside a range. For the code generation phase, only the knowledge of node extensibility is required. If the node stands for a rule, it is extensible and ready to be extended with sub-rules, otherwise, no action is performed to this node until the end.

2.2 Adding control to grammar rules

The generated code should contain a diversity of grammar structures and runtime class interactions, while being easy to mutate and free of non-interesting code structure. Simultaneously, the generated code must be compilable. To satisfy these requirements, two controls are added to the production of AS code during opportune stages of *ScriptGene* before compilation. First control consists of modifications to the grammar file and the second control consists of adjustments to the context during the mutation phase (Section 2.5). Although adjustments of the grammar file are less flexible than that of the runtime class mutations Python code, the foremost occurrence of the grammar file in the *ScriptGene* generation process necessitates that it be given a higher adjustment priority. This allows the removal of root causes of potentially troublesome structures from the compiler. G2 is therefore created from G1 under a few controls (e.g., simplification, ignorance, modification...) of select rules. G2 is then transformed into a Python source file by the ANTLR parser.

Simplification rules ANTLR allows Java code to be inserted into grammar rules to analyze the complex context of the source code. However, this kind of Java code behaves as comments to the ANTLR language parser, and is ill-suited for generating code snippets. A clean-up simplification of this Java code is required to facilitate the process of parsing G2 into Python format.

Ignorance rules A rule such as *xmlPrimaryExpression* is used to parse XML expressions, which are uninteresting to us, since we focus on the grammar structures of the context and not on particular API functions. Code generation cycles will be wasted if these rules are left unmodified. Thus, these rules' sub-rules are replaced with short XML expression instances.

Modification rules A rule such as *functionBody* is rewritten with a one-time loop structure. All the functions generated will start from one-time for-loops. This allows most breaks and continues inside those functions compiler-legal statuses.

Expansion and replacement rules Consider the rule *tryStatement* as an example. The main part of this rule is embedded inside a block, whose expansion according to our algorithm can simply be a random pick from its sub-atoms (see Section 2.3.2). Initialization of *tryStatement* sub-rule selection will not work. Therefore, depth-first sub-rule selection is disabled in this situation. The rule is rewriten and the block inside the *tryStatement* is replaced with a newly defined rule *tryStatementBlock*. Initialization of sub-rule selection on *tryStatementBlock* will work and give the code generation phase a greater diversity.

```
1: tryStatement
   : TRY blockStatement
     ( catchClause+ finallyClause
     | catchClause+
     | finallyClause
     )
   ;
2: tryStatement
   : TRY blockStatement tryStatementBlock
   ;
   tryStatementBlock
   : catchClause+ finallyClause
   | catchClause+
   | finallyClause
   ;
```

Marking of rules related to functions and variable declarations This is done for the runtime class mutation phase, to reduce the complexity of lexical analysis subsequently. For example, if we attempt to expand the *variableDeclaration* rule during the code generation phase, the sophisticated variable expression will be very difficult for the mutation algorithm to control both the type and the value of the variable. Therefore, its sub-rules are replaced with markers such that we can directly alter the type and the value of the variable.

2.3 Grammar-based blackbox fuzzing

According to vulnerability reports of AVMs in the past 2 years [20], many vulnerabilities were caused by AS code containing interactions with runtime classes. Therefore, a main aspect of *ScriptGene* is to create opportunities for interactions between different runtime classes.

2.3.1 Key points of the design

There are three key points that should be taken into consideration while creating compiler-pass AS code to test a backend AVM.

Stricter environment For a JavaScript-like VM, even if the backend VM receives no executable code due to the inappropriate grammar context of the input code, it is still possible to test the frontend parser and trigger bugs in trusted environments. Unlike JavaScript code and its VM, AS code interfaces with the AVM via a compiler. Bugs of the parser (in the compiler) would not be a direct threat to the trusted environment. Accordingly, AS requires much stricter context than JavaScript, since any preceding line of invalid AS code will block the execution of the remaining code due to the compiler. I.e., AS should be global-valid while JavaScript needs to be only line-valid. Adding controls to the code generation phase and marking rules inside G2 (see Section 2.2) overcomes this difficulty.

Additional grammar structures in AVMs In order to compile and execute code on the AVM, some additional structures are needed in our AS code. Such structures are difficult to generate with random walks over grammar

space, thus a brute force approach is inefficient. Therefore, code templates (see Section 2.5.2) are used to reconstruct the source code just one step before compilation.

Valid context Source files with proper structure and grammar still contain undefined variables, functions, classes, etc., and are thus still uncompilable. This is addressed by mutating the source code during the mutation phase (see Section 2.5) with real classes, API functions and predefined variables that are extracted from runtime libraries of the AVM.

2.3.2 *Generation of nearly-valid code snippets*

One possible approach for fuzzing AVMs based on grammar involves randomly generating source code that follow the grammar structure. However, naïve random walks over grammar rules are not guaranteed to terminate. Referring to the idea of *LangFuzz* [19], we bring an end to the random walk code generation process using instances of grammar rules learnt from known test suites. This is a good start, but in contrast to *LangFuzz*, it is not the final step before code execution. Furthermore, we enhance *LangFuzz*'s idea by incorporating both depth-first and breadth-first algorithms into *ScriptGene*, as described in algorithm 1. The

Algorithm 1 ActionScript code generation

Input:
A decision of sub-rule selection, *seednum*;
Output:
Nearly-valid code snippets, *codearray*;
1: Initialize *codearray* as [*startrulenode*];
2: Initialize the sub-rule selection based on *seednum* using a deterministic-random algorithm;
3: **while** (Number of active nodes and number of generation cycles are still low) **do**
4: **for** each *node* in *codearray* **do**
5: **if** *node* is not an active node **then**
6: Remain *node* unchanged;
7: **else**
8: Replace *node* with expanded nodes using breadth-first/depth-first algorithms of sub-rule selections;
9: **end if**
10: **end for**
11: **end while**
12: **for** each *node* in *codearray* **do**
13: Replace *node* with rule instances of its type;
14: **end for**
15: **return** *codearray*;

AS code is generated on a per-cycle basis. Each cycle of generation expands the nodes inside the *codearray*. Iteration is used instead of recursion, since determination of possible code length and number of nodes are more convenient with iteration. These important parameters control the termination of the code generation phase.

Each level of expansion is done on active nodes. The type of the node determines if a node is active or not. In figures 2, 4 and 3, each circle represents a node, while the character inside states the content. A given node is a NULL atom if the node is blank (previous level of expansion has randomly chosen zero for the quantifiers or optional). The quantifiers ('*','+') and optional ('?') are stored without any simplifications and will be replaced by a random number in its valid range during code generation. This is also different from *LangFuzz* [19]. Before a level of expansion, the nodes in the *codearray* are scanned and a hierarchy is established for each cycle: rule->block->atom. During each cycle, only the highest hierarchy item is set active. This ordered expansion is more effective and intuitive when we need to examine the generation process. Both ordered and non-ordered expansion processes are illustrated in figures 2 and 4 for comparison. In a non-ordered expansion process, such as illustrated in figure 2, different structures will inevitably begin to embed each other (a block inside a block, an atom inside a block embedded further inside a rule, extended from an atom...). Additionally, expansion cycles may be wasted on extending structures that will eventually become NULL (the left branch of figure 2, for example). Therefore, the ordered process as depicted in figure 4 is our node expansion approach. Although figure 4 appears to

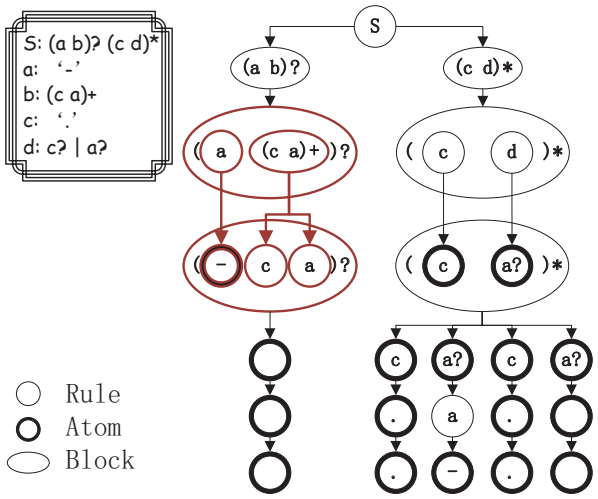

Figure 2: Node expansions in random order.

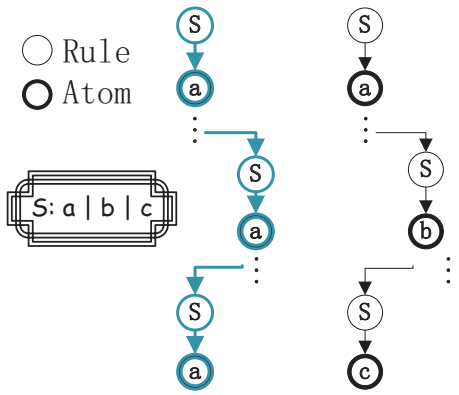

Figure 3: Left branch represents a depth-first sub-rule selection. Right branch represents a breadth-first sub-rule selection.

depict a breadth-first algorithm, it is not. Breadth-first or depth-first algorithms apply to the selection of sub-rules. Rules consist of sub-rules, separated by "|" in ANTLR such as "Rule S: a | b | c". Rule "S" can be selected as "a", "b"

or "c". Our breadth-first method is a random selection among sub-rules, whereas our depth-first method holds on to the initial selection at the beginning of the code generation phase (see line 2 of algorithm 1). Figure 3 demonstrates the difference between breadth-first and depth-first algorithms. Our breadth-first method is similar to that used by [19]. Depth-first algorithm was added, based on results of our previous testings of the regular expression interpreter [31]. This addition enables us to test AVMs with nested grammar structures. The idea is akin to sending a "((((((expr))))))", to test the expression interpreter. Nested structures appear to be validated with less caution and return more bugs. Intuitively, to cover more grammar rules, breadth-first algorithms need longer code lengths, while depth-first algorithms need more generation cycles with different sub-rule initializations. To terminate the code generation process as in line

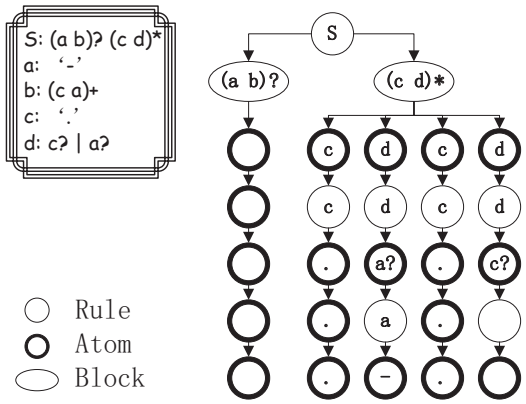

Figure 4: Node expansions following the rule, block, atom order.

13 of algorithm 1, we study code fragments (which are essentially examples for non-terminals in grammar rules and in our case, are instances of rules) from the AVM test suite in Tamarin [12], which is an open-source AVM project under Mozilla. The AS source code is taken from their acceptance test cases and dissected into pieces then categorized based on their rule types. A sample code fragment pool is built after parsing the test suite with the help of G1, in which there are several instances for each rule. The G1 grammar file originally belongs to a plug-in for Eclipse, intended to be part of a syntax highlighter function. It is imperfect when handling large assortments of code fragments. Since some parsed results may contain faults, the pool requires manual fixes. In addition, G2 is a modified version of G1, not all of the rules find their instances from this test suite. Thus, the missing ones are added manually.

2.4 Extraction of runtime information

After creating nearly-valid AS code snippets, all the identifiers pointing to labels, functions, variables and types are marked with the AS lexier. Some of these will be replaced with runtime classes, properties and API functions of the AVM, while others need modifications to suit the current context. A knowledge base is built for this purpose, containing the relevant runtime class information of the AVM.

Runtime class information can be obtained from mainly two sources. The first is the ActionsPanel_3.xml, found in the sub-directory of the SWF Integrated Development Environment (IDE), Adobe Flash Professional CS6 [4]. It contains sufficient information for the construction of expressions that declare variables to be instances of classes, call their member functions and modify their properties. However, it does not contain newly published APIs and classes.

The second one is the file playerglobal.swc, which accompanies newly released versions of Adobe Flash. This file contains the newly introduced APIs and runtime classes, and constitutes sufficient information for a compiler to determine if a class or function exists. By unpacking the playerglobal.swc file yields the files catalog.xml and library.swf. Inside catalog.xml are names of runtime classes, especially some undocumented API functions. Demonstrations of functions and properties are missing compared to ActionsPanel_3.xml. It is possible that they reside in library.swf in binary format. Instead of dealing with library.swf, the dynamic reflection mechanism of Flash is used to extract the missing information. Thus, *describeType* [1] is employed to restore the structure of the class using only its name.

After runtime information has been collected, verification is required for runtime class mutations, to ensure that they are compatible with our compilation environment. Our compiler asc.jar comes from Flex SDK [2], an open-source SDK designed to build Flash. Asc.jar performs compilation-checks by utilizing the information in playerglobal.abc. However, playerglobal.abc is not as up-to-date as playerglobal.swc. Therefore, the compilation and execution status of each class collected in the current AVM of interest is manually verified, to minimize the amount of non-executable test cases produced by the mutation phase.

To verify the compilation status of these runtime classes, an AS source file is built by importing all the classes collected, then called upon with *describeType* inside a "try-catch" clause. The classes that are not recognized (cause exceptions) by asc.jar and the target AVM are excluded, then the valid description output by *describeType* are recorded. For this purpose, a Python module is created that parses all the XML outputs of both *describeType* and ActionsPanel_3.xml, to build a runtime class pool in memory. This pool contains:

- name of runtime classes and their parent classes,
- name of member functions and the type of each parameter,
- name of dynamic and static properties of each class.

2.5 Runtime class mutation

This phase is a critical part of *ScriptGene*. Outputs of this phase are valid AS code that will be compiled into SWF files. The guiding idea of runtime class mutation is to bring more interactions between different classes and give a valid context to the code simultaneously. This is done in the following two steps.

2.5.1 Step 1, mark identifiers

We mark the identifiers pointing to names and types of functions, classes, catch variables, break variables, continue variables, labels and function-calls via lexical analysis. The lexier is based on G1, which recognizes all the identifiers. The identifiers are distinguished according to the characters of the former and latter tokens in the lexical sequence. For example, if a "while" clause is generated in code generation phase, then identifiers such as "DateCase" are neither API functions nor static classes and are not yet acceptable to the compiler. Subsequently, the nearly-valid AS source code

will become a marked file with all the identifiers replaced by markers, for example:

```
1: while(new Object+=a,
   DateCase.setMilliseconds(newms)||c,
   i++||f||2);
2: while(new _varname_+=_varname_,
   _varname_._funcall_(_varname_)||_varname_,
   _varname_++||_varname_||2);
```

2.5.2 Step 2, replace markers

The input of this step is the output of the previous step, an AS source code file full of markers. Our goal is to mutate these markers with real classes, build a context for each variable that will be used and to correct lexical issues to compensate for G1. For example, marked AS code will replace "_userdefine_" in the following prebuilt template:

```
package{
    import flash.display.*;
    _import_
    _globalvardeclaration_
    _userdefine_
    public class Main extends MovieClip{
        public function Main():void{
            _functioncall_
        }
    }
}
```

When the mutation begins, the input will be a template file full of markers. All the classes, API functions, global variables and member properties will be enumerated from the runtime class information pool as built in Section 2.4. Every mutation cycle will commence by picking several classes from the runtime class information pool. Presently, we have adopted a limit of two classes per cycle. During this process, all markers in the template file will be replaced with meaningful code.

"_import_" will be replaced with "import" clause to include the class of our choice. "_globalvardeclaration_" will be replaced with declarations of class objects with "new" clauses. References are made to the runtime class information pool to check the existence of constructors. If so, a "new" clause is built for this class with appropriate parameters. Otherwise, the chosen class is a static class. Properties and member functions of such a class can be used directly without declaration.

Next, a variable pool and a function pool are initialized. The variable pool and the function pool stores the variable names and methods of runtime class objects of our choice respectively, including all of their properties and their parents'. These two pools will be referred to, when markers referencing variables and function-calls are encountered. In addition, there is one more global pool to record the classes that *ScriptGene* generates and their variables, functions and labels.

When *ScriptGene* scans the marked template file, it will encounter several kinds of markers. Markers referring to variables will be replaced as listed below.

- A variable pointing to the instance of a picked class.
- A property of the chosen class and its parent.
- A member function of the chosen class and its parent. Member functions can be seen as variables in AS under specific circumstances. Thus, they can be incorporated into grammar structures, such as assignment expressions.
- A property or member function of the chosen class in prototype form. This is added, since prototype is a way to inherit for this object-oriented program. We expect that it will lead to more interactions between different classes.

Under special circumstances, only pure variables are permitted (variables that are instances of classes), such as "_varname_++", "--_varname_". These situations are identified and replaced with only predefined instances of the chosen classes.

The function-call mutation is similar to the variable mutation. All the markers pointing to function-calls with member functions of chosen classes and their parents are replaced appropriately.

Other than the runtime class mutations, grammar structures in the marked template file also needs manual fixes. Some are fixed by altering G2 in Section 2.2, such as "continue" clauses. To accomplish this, we maintain a label pool to record recently declared labels. Once "continue" clauses are encountered that request for labels, the labels are substituted and appended. Although this process is not foolproof, it is necessary to enable a substantial portion of the code to compile. Other fixes on grammar structures are too trivial to be listed here and can be found on our website [11].

When all the variables and function-calls used in the code snippets have been declared, the instruction flow is altered to be logical and grammar errors are fixed. The "_functioncall_" markers are replaced by calls to member functions that *ScriptGene* has generated and mutated. Then, all the classes from the global pool are initialized and every function is called to enable our generated code to run in the target AVM. This completes the generation of compilable AS source code.

3. EVALUATION

Two experiments were conducted with *ScriptGene*. The first evaluates different generation and mutation strategies: breadth-first or depth-first during code generation phase and multiple or single templates during runtime class mutation phase.

The second compares our code coverage with the original Tamarin test suite. The results show that *ScriptGene* achieves a much better code coverage than Tamarin in all three versions of AVMs tested. A brief analysis of typical bugs found during the second experiment and details about how they were found will be given.

3.1 Testing conditions

The testing process consists of three parts: a generation phase, a runtime class mutation phase and an execution phase. The first part typically requires about half a day on a PC (equipped with an Intel Core Q9400 with 4GB of DDR3 RAM), to generate about ten thousand source code files with distinct sub-rule initializations and transform them to marked template files with lexical analysis. The mutation phase is then given a week of computation time. Millions of SWF files are produced and stored during this phase. Finally, all the SWF files are executed in the AVM.

To sequentially send test cases to the AVM via Adobe

Flash ActiveX Control, a wrapper was constructed and employed. When bugs occur that result in non-severe memory leaks, the AVM may not crash directly or immediately. Therefore, all test cases are sent by the wrapper in the same process, which amplifies memory leak effects on the AVM by accumulating multiple persistent memory leaks. This can either directly cause or hasten AVM crashes. Each SWF file is allowed at most one second of execution time before they are freed from memory.

Since we lack the source code of the AVMs tested, we resort to block coverage instead of line coverage to evaluate our fuzzing performance. To record the block coverage, we refer to the idea of PaiMei [24] and use PIDA to analyze the Adobe Flash ActiveX Control by recording the start and end addresses of each block inside. Basic block coverage is the same as statement coverage, except that the unit of code measured is each sequence of non-branching statements. A block is a basic element of the disassembled code in IDA Pro [8]. Once the start address is covered, all the assembled code in this block will be executed.

The debugger we have built is based on Pydbg (a subproject of PaiMei), to record block coverages and memory corruptions caused by the test cases. Once the wrapper begins loading the test cases, the debugger is attached and the breakpoints on basic blocks over Adobe Flash ActiveX Control are set. If any of the breakpoints are triggered, it is removed from the list for the remainder of the test. Subsequently, all the breakpoints that have been triggered (to record the total block coverage) during the execution phase are collected. If the AVM crashes due to some test cases, the debugger will record the current crash information and the test case file for further analysis.

There are two additional reasons for running the test cases in a single process. First, setting breakpoints on the wrapper requires no more than a few minutes, but are only done whenever the wrapper is restarted (e.g., such as due to crash-related wrapper terminations). Running the test cases in a single process instead of a new process for each test cases saves time spent on setting breakpoints. Second, in a multiple processes approach, it is possible to miss bugs that are only triggered together by multiple test cases. The root causes of these types of bugs are hard to locate in a multiple-test case, multi-processes scenario.

All the experiments are done through three versions of AVMs. Since many features of AS3 are not available until Adobe Flash 10, our first choice is version 10.0.45.2. This is the very first version of Adobe Flash 10. The second version, version 10.2.152.32 is chosen from the period when AVM's vulnerabilities began to attract security engineers' attention, around the time of CVE-2010-1297 [20]. The last version is the newest version available during our tests, version 11.3.300.265.

3.2 Comparison of code generation and mutation methodology combinations

In total, there are four code generation and mutation methodologies: breadth-first or depth-first methods for code generation, single or multiple templates methods for runtime class mutations. The first two methodologies that govern node-order expansions are given in Section 2.3.

The other methodologies regard to varying initial templates by choosing single or multiple templates during the mutation phase. For the single template methodology, an initialized selection of a single marked template file is used for the mutation phase. For the multiple templates methodology, each batch of mutations is done on a candidate selection, chosen randomly from all available marked template files. The multiple templates methodology is capable of more initialization choices of sub-rule selections when compared to the single template methodology. Four unique combinations of these methodologies are: breadth-first&single template (BS), breadth-first&multiple templates (BM), depth-first&single template (DS) and depth-first&multiple templates (DM).

Version	10.0.45.2	10.2.152.32	11.3.300.265
Blocks	15206	21819	34532
BM	4635(30.48%)	4330(19.85%)	5909(17.11%)
BS	3691(24.47%)	5152(23.61%)	5703(16.52%)
DM	3477(22.87%)	4660(21.36%)	7549(21.86%)
DS	2350(15.45%)	4353(19.95%)	4770(13.81%)

Table 1: Block coverages of different combinations of methodologies. The amount of covered blocks and the percentages of total blocks covered are tabulated.

Each combination is allotted a week of computation time for generation and mutations. The output of each is about $\sim 800,000$ SWF files. Then, an additional one to two weeks is required for execution, when the triggered breakpoints are recorded to calculate block coverage (see table 1).

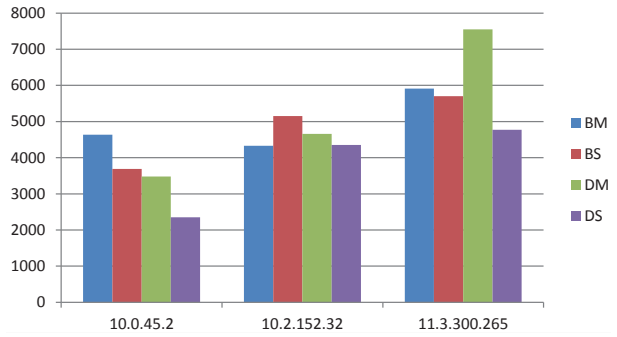

Figure 5: Block coverages under different versions of AVMs.

As seen from figure 5, the best combination in terms of coverage is not obvious, while the worst is the DS combination. During the tests on version 10.2.152.32, some test cases inside the BM combination have led to a memory bug of the AVM. Subsequently, the AVM entered a self-protection state that prevents further execution of a variety of SWF files. This explains the sudden fall of the BM combination's block coverage. For version 11.3.300.265, the DM combination has a greater increase in coverage as compared to the other combinations and the former results. This is because BM has randomly picked several marked template files containing more grammar structures than the other combinations. In fact, the DM and BM combinations are the most representative of ideal situations, where the diversity of initial templates is maximized, combined with extensive runtime class mutations. In reality, however, close approximations to this ideal results in a very resource-intensive process. Therefore, prior to fuzz testings, considerations should be made

regarding the amount of computational resources available and choose methods accordingly.

These combinations are cross-compared in table 2. Each item contains three parts: blocks in common that have been covered during the tests and the proportion of these blocks to the total blocks. For example, the DM and DS cross-comparison under version 11.3.300.265 includes 4770 blocks that both combinations covered. These 4770 blocks represent 100% of the DS combination's coverage, and only 63.2% of the DM combination's coverage. This suggests that the DS combination is less useful for testing version 11.3.300.265, as compared with the DM combination.

Version	10.0.45.2	10.2.152.32	11.3.300.265
BM,BS	3681 79.42%,99.73%	4098 94.64%,79.54%	5172 87.53%,90.69%
DM,DS	2329 66.98%,99.11%	3770 80.90%,86.61%	4770 63.19%,100.0%
BM,DM	3454 74.52%,99.34%	3959 91.43%,84.96%	5773 97.70%,76.47%
BS,DS	2322 62.91%,98.81%	4350 84.43%,99.93%	4769 83.62%,99.98%

Table 2: Cross-comparison between different combinations of methodologies.

To summarize, when the testing situation involves limited computational resources and/or time, the DS combination is inappropriate. The multiple templates methodologies seems to be more satisfactory than the single template methodologies. When the template methodology is fixed, the performance of the overall combination seems to depend on whether breadth-first or depth-first algorithms performs better on the particular AVM (e.g., the BM combination is better than the DM combination for version 10.0.45.2, and worse for version 11.3.300.265).

3.3 Comparisons between ScriptGene and Tamarin

The open source AVM implementation Tamarin contains acceptance test cases [3], which can be seen as the official test suite of AVMs and is employed as instances of grammar structures in the generation phase of *ScriptGene*. Block coverage comparisons between the test cases of Tamarin and *ScriptGene* should be a direct proof that *ScriptGene* is an effective approach to test AVMs. Normally, the test suite

Version	10.0.45.2	10.2.152.32	11.3.300.265
S	4667(30.69%)	5727(26.25%)	7697(22.29%)
T	2474(16.27%)	3351(15.36%)	3686(10.67%)
S,T	2408 51.60%,97.33%	3236 56.50%,96.57%	3622 47.06%,98.26%

Table 3: Block coverages and cross-comparison between *ScriptGene* (S) and Tamarin (T).

of Tamarin consists of AS source code and can only be built as ".abc" files that are compatible with the Tamarin Virtual Machine. However, thanks to Tamarin's support, a solution [10] has been created to compile SWF files with Tamarin-AS source code under cygwin [5]. There are 603 SWF files of Tamarin acceptance test cases. Using the same method as in Section 3.2 the block coverages of Tamarin test suite are recorded in the three versions of AVMs. The SWF test

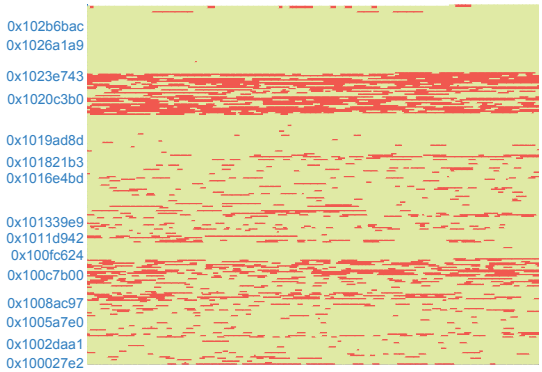

Figure 6: Block coverage of Tamarin.

cases generated by *ScriptGene* are the same as in the first experiment. Triggered blocks are merged among the different combinations to give us a net total block coverage of *ScriptGene*.

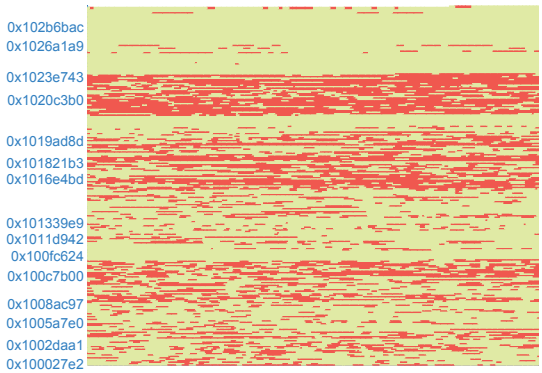

Figure 7: Block coverage of *ScriptGene*.

From table 3, it is seen that *ScriptGene* can cover most blocks (>96%) that are covered by Tamarin test cases in the three versions of AVMs. Vivid squares in figure 6 and figure 7 are used to demonstrate block coverages. Red strips represent covered blocks and dark yellow represents untouched blocks.

All the blocks are arranged according to their real memory addresses and lengths. The base address of Flash ActiveX DLL is 0x10000000, located at the bottom of the figures. Blocks with higher addresses are closer to the top. The figures look similar for the three versions tested. Therefore only one version is shown here. Figures 6,7,8 are the test results of version 10.0.45.2. The highest concentration of blocks covered by the Tamarin test suite lies near the top quarter of the memory space, which mainly deals with grammar and interpretation of the AS code. Blocks in lower addresses are those dealing with the SWF format and external resources of the operating system.

In figure 8, the blocks that was neither triggered by Tamarin nor *ScriptGene* are excluded. Remaining blocks are covered

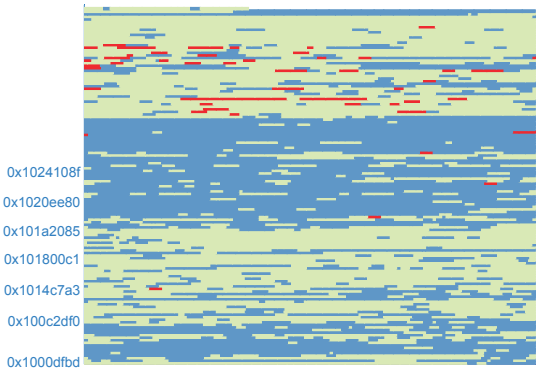

Figure 8: Cross-comparison between *ScriptGene* and Tamarin.

by only Tamarin (in red), only *ScriptGene* (in blue) or both Tamarin and *ScriptGene* (in light green). The improvement of block coverage by *ScriptGene* is seen to be very significant. The complexity of grammar structures produced by *ScriptGene* is certainly not less than that offered by the Tamarin test suite. In addition, with runtime class mutations, *ScriptGene* is able to create more interactions between various structures. We reason that both of these aspects have enabled us to achieve the higher block coverage.

3.4 Examples of bugs

Next, we would like to explain the generation process of the test cases that triggered some AVM bugs. These bugs were found by *ScriptGene* and missed by Tamarin. The original test cases can be found on our website [11]. Because our test cases contain many lines and each line consists of multiple structures of AS instructions, it is not immediately obvious which line or structure triggered a specific bug. Thus, for the following passages of this section, only the essential parts pertaining to each bug are maintained and simplified. *ScriptGene* found four bugs in version 10.0.45.2, one in version 10.2.152.32 and one in version 11.3.300.265. Although these bugs do not exist in the newest version of AVM at the time of writing, we have not yet found any published documentations on them. It is possible that Adobe adjusted the code of newer versions of AVMs, without reporting these bugs. Of the six bugs, three will be briefly detailed here, to illustrate the effectiveness of our strategies in code generation, runtime class mutation and testing.

3.4.1 Complex grammar structure generation

```
1: for each(var o in new <Object>
   [1,2,3,"hello",'out',"there",true,false,3.14159])
   {str+=o;}
2: for each(var _vardeclarationname_ in new<_varname_>
   [1,2,3,"hello",'out',"there",true,false,3.14159])
   {_varname_+=_varname_;}
3: for each(var var2:Number in new <var2>
   [1,2,3,"hello",'out',"there",true,false,3.14159])
   {var2+=var2;}
```

A bug leading to memory corruption is caused by malformed enumerations against the *vector* type. A *number* variable is made to point to a *string* under such code grammar, which leads to the AVM accessing unexpected memory. The ability to find this bug depends on the generation of valid code that are sufficiently grammatically-complex. As an example of the process of how this type of code snippet is generated, consider the following. First, the code generation phase outputs an AS code file containing a "for-loop" structure. Second, lexical analysis replaces the identifier with markers. Finally, in the mutation phase, "_vardeclarationname_" will be replaced with "varX:vartype" (where "X" is a number depending on the order of this pure variable and "vartype" will be chosen randomly from runtime class). "_varname_" will be replaced by a random pick from the variable pool (for instance, "varX").

3.4.2 Mutate with prototype

```
1: switch(actualmatch=string.match(pattern))
2: switch(_varname_=_varname_._funcall_(_varname_))
3: switch(
   var3.constructor=var3.append(
   flash.geom.Matrix3D.prototype.append
   ))
```

Member function *append* could cause memory corruptions when its parameter contains the prototype of this class. During the code generation phase of *ScriptGene*, a "switch" structure is built. Then, the identifier is marked and replaced by member functions, properties and instances of "flash.geom.Matrix3D" during runtime class mutations. This definitively supports the idea about bringing prototypes into the source code files, in order to allow more interactions between different classes as mentioned in Section 2.5.2.

3.4.3 Running tests cases in a single process

Another bug is found due to the method used to run the test cases. As mentioned before in Section 3.1, the wrapper of Flash ActiveX control loads the test cases sequentially in a single process.

In version 10.0.45.2, it was found that many access violations are triggered during the testing phase. Our debugger records the files that have caused the access violations. Attempts to reproduce the corruptions by sending the recorded test cases directly into the AVM results in no access violations. This implies that, some test cases prior to the recorded ones are probably also responsible for the access violations. Typically, this is around 2000-5000 test cases prior to the recorded test case. After several attempts to reproduce the corruptions, it is found that only the quantity of the test cases is important. If $\sim$(30-50) test cases are loaded simultaneously with none being freed from memory, the corruption will be reproduced, irrespective of the content in the test cases.

Tests on version 11.3.300.265 found another bug similar to the bug of version 10.0.45.2. However, for this version, the required test cases increased to 40000-120000 to trigger the corruption. Our attempts to reproduce and analyze the bug with the same method used on version 10.0.45.2 was not fruitful, since the newer AVM takes counter-measures against unbounded memory allocations. When the AVM detects that the following operations can potentially exhaust the memory, it terminates the AVM immediately and produce a big exclamation mark in the Flash GUI. This protection mechanism complicates the simplification and dis-

tillation of the test cases. Thus, we are uncertain if these access violations are due to the same reason as in version 10.0.45.2. It is possible that these memory leak problems have existed in Flash for many versions. Each subsequent revision reduced the effect of memory leaks somewhat, possibly through counter-measures, such that newer versions gradually required more test cases to trigger the memory leaks.

4. RELATED WORK

Fuzz testing was introduced in 1972 by Purdom [25]. Purdom used a syntax-directed method to generate test sentences for a parser. He gave an efficient algorithm for generating short sentences from a context-free grammar such that each production of the grammar was used at least once and tested LR(1) parsers using this technique. It is one of the first attempts to automatically test a parser using its grammatical structure. In 1990, Miller et al. [21] were among the first to apply fuzz testing to real world applications. In their study, the authors used randomly generated program inputs to test various UNIX utilities. Since then, the technique of fuzz testing has been used in many different areas, such as protocol testing [13, 26], file format testing [28, 29], or mutations of valid inputs [22, 29].

The most relevant studies for this paper are recent ones on grammar-based fuzzing and test generations for compilers and interpreters. In 2011, Yang et al. [32] presented *CSmith*, a language-specific fuzzer operating on the C programming language grammar. *CSmith* is a pure generator-based fuzzer, generating C programs for testing compilers and is based on earlier works of the same authors and on the random C program generator published by Turner [30]. Drawing a parallel to our work, they have used the built-in grammar to create compilable programs. Furthermore, they introduced semantic rules during their generation process by using filter functions, which allow or disallow certain productions depending on the context. In contrast to our work, *ScriptGene* takes less control over the instruction flow (we only restrict the behaviour of breaks and continues) during the generation phase, and leave the compiler-pass task to the runtime class mutation phase. This approach allows us to generate a greater diversity of grammar structures during the generation phase.

In 2012, Holler at al. [19] proposed *LangFuzz*, a grammar-based fuzzing framework that has been proven to be versatile in discovering vulnerabilities in JavaScript engines. We especially adapted *LangFuzz*'s idea of the "Shortest Terminal String Algorithm" for *ScriptGene*. Similar to *LangFuzz*, we have used grammar instances extracted from a test suite of a certain language to terminate the code generation branch. In contrast, *LangFuzz* mainly deals with JavaScript that requires only line-validity, while *ScriptGene* must maintain full-text validation with the help of modified grammar rules and code templates. Furthermore, *ScriptGene* adds runtime class mutations, legalizes the code and covers more branches in tested AVMs simultaneously.

As shown by Godefroid et al. [17] in 2008, a grammar-based fuzzing framework that produces JavaScript test cases can increase coverage when linked to a constraint solver and coverage measurement tools. They present a dynamic test generation algorithm where symbolic execution directly generates grammar-based constraints whose satisfiability is verified using a custom grammar-based constraint solver. Unfortunately their work in its current form only suits well on JavaScript VMs, since the feedback adjusts the input directly. However, AVMs only accept bytecodes, which are the outputs of the compiler. This additional layer prevents the direct feedback, rendering the symbolic execution ineffective.

Fuzzing web browsers and their components is a promising field, particularly in the case of Adobe Flash, which are also used in some hybrid documents. To date, several approaches have been taken to find bugs/vulnerabilities in Flash.

Flash as a particular format of documents could be fuzzed using a variety of file format fuzzing frameworks such as Peach [9] and SPIKE [14]. One of the methods of file format fuzzing called Dumb Fuzz bitflips every bit in the entire file. Despite its name, Dumb Fuzz was found to be able to test AVMs to a certain extent. Since AS code is stored as bytecodes in a SWF, single bit change in these bytecodes would in general completely alter the original meaning of the AS code. Several vulnerabilities were confirmed to be found by Dumb Fuzz recently [20]. Most of these bugs are due to type confusion. Late in 2011, Google used large scale computing resources to fuzz AVMs, relying solely on Dumb Fuzz and found tens of bugs [18]. Nevertheless, this type of method is mutation based and highly dependant on the source of the parent SWF. The source for Google was 20TB of downloaded SWF files. While 20TB cannot possibly include all the structures of AS, mutations on the bytecode level is a good way to explore additional code paths of AVMs by bypassing the compiler and alter the inputs directly, when computational resources permit.

There are other fuzzing approaches targeting Flash. However, most of them mainly focus on whether AS functions give enough validation to their parameters. In our previous work [31], we have attempted to fuzz the Regular Expression functions of AS. Using grammar-based mutation fuzzing, we have constructed much more sophisticated expressions to test the Regular Expression interpreter of AVMs, where several vulnerabilities were subsequently found.

5. DISCUSSION

Approximate grammars Other than the additional controls to the AS grammar rules as mentioned in Section 2.2, several other structures have been simplified. Package => Class => Function => Statement is essentially the generated code in a macro-view. Serialization of Package => Function, Class => Expression... is impossible due to the modifications of the grammar rules. Fixed serialization of code structures helps us keep track of the properties and functions and enables us to build them valid contexts. We need more domain knowledge of a particular language and its grammar, if we want to build a valid context for all complex grammar rules.

Resource limitations For the code generation phase, only several hours was needed to generate around ten thousand nearly-valid code snippets with different initializations of sub-rule selections. During each batch of runtime class mutations, only two of the classes from the pool were chosen. About ten days of mutation time yields only ∼10% of the mutations possible under our current strategies. Considering all possible combinations and the depth of the grammar, these code snippets explore only a tiny subset of AVM execution paths, since the mutation phase requires far more time than the generation phase. This is practically inevitable

with the blackbox fuzzing method. Additional information from whitebox analysis should be helpful to constrain the possibilities for future researches. Distributed computing environments such as cluster computing or cloud computing would be helpful for larger mutation diversities.

Finding new vulnerabilities We have used the compiler to generate complex bytecodes from AS source code, then execute the resulting files on the AVMs. Our approach is seen to be effective in achieving a good code coverage, as demonstrated in Section 3. However, we have found less bugs than expected, from our comparatively greater code coverage (compared to the Tamarin official test suite). Nevertheless, this is reasonable, since the discovery of bugs from random fuzzing is probabilistic. Evaluating a given fuzzing approach by the amount of bugs found is not appropriate, unless computational resources permit a large and statistically valid amount of runs. Therefore, code coverage and comparisons are better indicators of the effectiveness of *ScriptGene*. The discovery of bugs should be considered secondary to code coverage testing. In our future works, we would like to add bytecode mutations to the test cases produced by *ScriptGene*. Based on current code coverage, we deduce that bytecode mutations will be capable of identifying even more bugs by extending the code coverage. Bytecode mutations would also be a more direct method to test AVMs, since it bypasses the compiler-check.

6. CONCLUSION

ScriptGene is a novel approach to fuzz testing ActionScript Virtual Machines, where compilers are involved. We have extended and expanded the ideas of *LangFuzz* from JavaScript to ActionScript, to generate code snippets, then produced nearly-valid ActionScript code with additional controls. Finally, using runtime class mutations, we produce grammatically-complex compilable code that are rich in runtime class interactions to ultimately test a few AVMs. Our evaluation shows that our approach, *ScriptGene* explores deeper execution paths and is capable of nearly twice the code coverage compared to official tests (Tamarin). Our code coverage and discovery of unreported bugs found by *ScriptGene* in three different versions of AVMs demonstrate the effectiveness, validity and novelty of our approach.

7. ACKNOWLEDGMENTS

We thank Steven Ergong Zhang from University of Ottawa for his support and suggestions. This work is supported by National Natural Science Foundation of China (NSFC) under Grant 60970140, Beijing Natural Science Foundation (4122089) and China Postdoctoral Science Foundation Project (2011M500416, 2012T50152).

8. REFERENCES

[1] `http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/utils/package.html#describeType()`.

[2] About flex sdk. `http://sourceforge.net/adobe/flexsdk/wiki/About/`.

[3] Actionscript acceptance tests. `https://developer.mozilla.org/en-US/docs/Tamarin/Tamarin_Acceptance_Testing/Actionscript_Acceptance_Tests`.

[4] Adobe flash professional cs6. `http://www.adobe.com/en/products/flash.html`.

[5] Cygwin. `http://www.cygwin.com`.

[6] Ecmascript. `http://www.adobe.com/devnet/actionscript/articles/actionscript3_overview.html`.

[7] Flex formatter. `http://flexformatter.cvs.sourceforge.net/viewvc/flexformatter/ActionscriptInfoCollector/ASCollector.g3?view=log`.

[8] Ida:about. `http://www.hex-rays.com/products/ida/index.shtml`.

[9] Peach fuzzing platform. `http://peachfuzz.sourceforge.net/`.

[10] Running tamarin acceptance tests. `https://developer.mozilla.org/en-US/docs/Tamarin/Tamarin_Acceptance_Testing/Running_Tamarin_acceptance_tests`.

[11] Scriptgene project. `http://www.nipc.org.cn/project/ScriptGene`.

[12] Tamarin. `https://developer.mozilla.org/en-US/docs/Tamarin`.

[13] D. Aitel. The advantages of block-based protocol analysis for security testing. *Immunity Inc., February*, 2002.

[14] D. Aitel. An introduction to spike, the fuzzer creation kit. *immunity inc. white paper*, 2004.

[15] B. Binde, R. McRee, and T. Oąŕ Connor. Assessing outbound traffic to uncover advanced persistent threat. *SANS Institute. Whitepaper*, 2011.

[16] D. Blazakis. Interpreter exploitation. In *Proceedings of the USENIX Workshop on Offensive Technologies*, 2010.

[17] P. Godefroid, A. Kiezun, and M. Levin. Grammar-based whitebox fuzzing. In *ACM SIGPLAN Notices*, volume 43, pages 206–215. ACM, 2008.

[18] Google. Fuzzing at scale, 2011. `http://googleonlinesecurity.blogspot.nl/2011/08/fuzzing-at-scale.html`.

[19] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 38–38, Berkeley, CA, USA, 2012. USENIX Association.

[20] H. Li. Understanding and exploiting flash actionscript vulnerabilities, 2011.

[21] B. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[22] P. Oehlert. Violating assumptions with fuzzing. *Security & Privacy, IEEE*, 3(2):58–62, 2005.

[23] T. Parr and R. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[24] pedramamini. Paimei. `http://pedramamini.com/PaiMei/docs/`.

[25] P. Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, 1972.

[26] G. Shu, Y. Hsu, and D. Lee. Detecting communication protocol security flaws by formal fuzz testing and machine learning. *Formal Techniques for Networked*

and Distributed Systems–FORTE 2008, pages 299–304, 2008.

[27] J. Smith and R. Nair. *Virtual machines: versatile platforms for systems and processes*. Morgan Kaufmann, 2005.

[28] M. Sutton and A. Greene. The art of file format fuzzing. In *Blackhat USA Conference*, 2005.

[29] M. Sutton, A. Greene, and P. Amini. *Fuzzing: brute force vulnerabilty discovery*. Addison-Wesley Professional, 2007.

[30] B. TURNER. Random c program generator, 2007. `http://sites.google.com/site/brturn2/randomcprogramgenerator`.

[31] D. Yang, Y. Zhang, and Q. Liu. Blendfuzz: A model-based framework for fuzz testing programs with grammatical inputs. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*, pages 1070–1076. IEEE, 2012.

[32] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. *ACM SIGPLAN Notices*, 47(6):283–294, 2012.