Experimental Analysis & Refinement of a Guided Exploit Generation Technique for Language Virtual Machines

Fadi Yilmaz *
fadiyilmaz@ybu.edu.tr
Ankara Yildirim Beyazit University
Dept. of Computer Engineering
Ankara, TURKEY

Meera Sridhar
msridhar@uncc.edu
UNC Charlotte
Dept. of Software and Info Systems
Charlotte, North Carolina

Wontae Choi †
wtchoi.kr@gmail.com

ABSTRACT

Background: We discussed GuidExp, a semi-automatic exploit generation tool, in our paper at ACSAC 2020. GuidExp generates an exploit script for a given ActionScript vulnerability. Unlike the other exploit generators, GuidExp does not use fuzzing or a symbolic execution; rather, it relies on human expertise to guide it in successfully discovering vulnerable execution paths. This paper augments our ACSAC paper and provides more details on the experiments we conducted.

Aim: We sought to show that GuidExp can generate working exploit scripts for real-world vulnerabilities in open- and closed-source ActionScript Virtual Machine (AVM) implementations.

Data: We used community artifacts as GuidExp inputs: a ROP gadget sequence, used to generate a GuidExp sub-goal; a proof-of-concept (PoC) exploit script for CVE-2015-5119; and vulnerable AVM implementations.

Method: We conducted a series of experiments, where the results of each informed the development of the next. First, we tested GuidExp on an open-source AVM using a single ActionScript vulnerability (CVE-2015-5119). We measured the number of candidate slices generated by GuidExp, the number of candidate slices executed successfully, and the time required to achieve each exploit subgoal. We then implemented GuidExp optimization techniques based on what we learned and repeated the same experiment using the optimizations. Next, we ran the experiment using eleven different vulnerabilities in a closed-source AVM. Finally, we ran the original experiment using a larger search space to understand how less accurate human guidance might impact GuidExp's performance.

Results: Using an optimal search space (expert human guidance), GuidExp generated a successful exploit script for CVE-2015-5119 in an open-sourced AVM in just over 14 minutes. When the search space was increased to by a factor of 2, the time increased to just over 59 hours. GuidExp generated successful exploit scripts for 11 vulnerabilities in closed-source Flash Player v11.2.202.262, with the longest completion time being just under 14 hours when using an optimal search space.

Conclusions: GuidExp needs to be human-guided to generate an exploit script. Its performance is greatly affected by the accuracy of the provided exploit subgoals. Redundant instructions for an exploit subgoal significantly increases the search space and subsequently, the time that GuidExp needs to generate an exploit script.

1 INTRODUCTION

Determining *exploitability* [57] of a given vulnerability has historically been a labor-intensive manual process requiring deep security knowledge. With the recent advances in fuzz testing and symbolic execution, several approaches for automatically generating exploits have been proposed [1, 11, 14, 15, 17–19, 22–26, 31, 34, 35, 41, 46, 50–56, 58]. These approaches, collectively known as *automatic exploit generation* (AEG), (such as AEG for *return-oriented programming*, or *control-flow hijacking*) have become critical tools for auditing software security, and attack prevention.

AEG implementations are usually driven by one of two engines: a *fuzzer* [36] and a *symbolic execution tool* [32]. The fuzzer helps explore the input-space by monitoring the execution of randomly generated inputs, and the symbolic execution tool helps explore the execution-path-space by symbolically executing every execution path. However, both approaches have their own limitations in the space of AEG for language *virtual machines* (VM).

Typical fuzz testing approaches do not scale well for applications taking as input other computer programs, such as language VMs. They do not efficiently generate inputs for such applications [28, 45]. For example, a fuzzer is extremely unlikely to test all possible behaviors of a program (e.g., the probability of executing the "then" branch of the if-statement `"if (x==3)"` is only $1/2^{32}$ assuming `x` is 32-bit `integer` value). While smart fuzz testing approaches [13, 33, 45] can generate random structured inputs (e.g., DNS packages), they cannot adopt complex grammar rules (e.g., generating valid program binaries with correct offsets). Thus, traditional fuzzers typically struggle to perform exploit generation for language virtual machines.

AEG implementations for language VMs also cannot utilize a typical symbolic execution tool due to its limitations. Symbolically executing a language VM raises the path-explosion problem in the early stage of the AEG process [12]. For example, the VM produces an execution branch for every instruction it can read during the parsing phase to obtain the sequence of instructions to be executed. Although the general purpose of path selection heuristics is to deal with execution-path space [12, 27, 31, 49], they are not immediately helpful as the number of execution branches that symbolic execution tools need to interpret is almost as many

*The work reported herein was performed while at UNC Charlotte.
†This work was done while Wontae Choi was employed at Google Inc. However, the work is a personal project and did not happen in the Google Inc. context. The work also does not express the views or opinions of Google Inc.

Learning from Authoritative Security Experiment Results (LASER) 2020
8 December 2020, Virtual
ISBN 1-891562-81-9
https://dx.doi.org/10.14722/laser-acsac.2020.23111
www.acsac.org

as all possible inputs that the language virtual machine can take. For this reason, an AEG implementation sometimes adopts a hybrid approach, switching between the two techniques [15] when one technique hits its limitations.

In our previous work [55], we presented GuidExp, the first *guided* (semi-automatic) exploit generation tool that does not rely on fuzzers or symbolic execution engines. While typical AEG implementations synthesize a whole exploit script whose execution path reaches one of predefined exploited program states, GuidExp leverages *exploit deconstruction*, a technique of splitting the execution path into many shorter paths, to reach the exploit program state. Hence, GuidExp can synthesize code snippets that follow these shorter paths. GuidExp expects that program states on which the execution path is split are given and described by a security expert as *exploit subgoals*.

GuidExp focuses on generating *Return-Oriented Programming* (ROP) [48] attack scripts, and demonstrate such an attack for an AVM vulnerability that we use in our experiments. GuidExp uses ROP attacks as representative attacks, because since 2015 almost 80% (547/698) of disclosed ActionScript (AS) vulnerabilities could lead to an arbitrary code execution by implementing an ROP attack [37]. Therefore, we ensure that GuidExp is expected and capable of generating exploit scripts that perform an ROP attacks. In an ROP attack, an attacker hijacks program control-flow by gaining control of the call stack and then executes carefully chosen machine instruction sequences that are already present in the machine's memory, called *gadgets* [16]. Each gadget typically ends with a `return` instruction that allows the attacker to craft an instruction chain that performs arbitrary operations. We want to highlight, however, that GuidExp can synthesize exploit scripts that perform any type of attack (not just ROP) for given vulnerabilities if the corresponding PoC and exploit subgoals are provided.

In this paper, we discuss experimental analyses and refinements we conducted as part of developing GuidExp. We introduce technical challenges we encountered during the implementation of GuidExp, and how we overcome these challenges. We hope these discussions are interesting to the reader since GuidExp is the first exploit generation tool that can synthesize ROP exploit scripts.

The main contributions of this paper are:

- We discuss our experimental analyses of our exploit generation technique for language VM vulnerabilities.
- We discuss how the result of initial experiments allows us to identify challenges of the exploit generation problem.
- We present our four optimization techniques and reasons why we need to adopt them.
- We present the performance numbers of GuidExp in our final experiments in which GuidExp exploits eleven AVM vulnerabilities.

The rest of the paper is organized as follows. Section 2 presents background on ROP attacks and our exploit generation approach. Section 3 presents the three phases of our exploit generation technique and introduces the main components of GuidExp—the Code Generator, the Invariant Validator, and the Exploit Subgoal Manager. Section 4 provides implementation details of GuidExp to provide the reader context for our experimental setup. Section 5 introduces the initial experimental setup and artifacts that we borrowed from the community. Section 6 introduces our optimization techniques. Section 7 presents our experiment refinements that led to better performance numbers and Section 8 concludes.

2 BACKGROUND

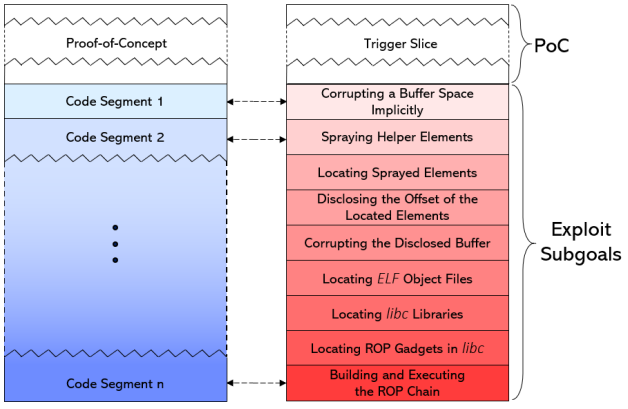
2.1 Structure of a Typical ROP Attack

In this section, we introduce the structure of a typical ROP attack. Fig. 1a depicts the structure of a typical ROP attack. An ROP attack starts with executing the PoC—the piece of code which triggers the vulnerability. The PoC corrupts the memory by performing activities such as creating a dangling pointer, or mangling the structure of the garbage collector. However, the execution of the PoC should not raise a *kernel panic* [20] (a system error from which operating systems cannot quickly or easily recover), because otherwise, the exploit that contains the PoC would result in the same kernel panic, and the operating system terminates the execution of the exploits before they perform their intended malicious activities. The ROP attack exploits the resulting corrupted memory that the execution of the PoC caused, and performs unauthorized activities on the memory until it builds a gadget chain performing the arbitrary operations. The ROP attack achieves its malicious end goal in several exploit subgoals, each subgoal which we demonstrate with Code Segment # in the Fig. 1a.

2.2 Intuition Behind Target Exploit Generation

In order to facilitate exploit generation, we define a structure for our *target exploit*, which is a high-level, semantic outline of the final exploit we expect GuidExp to generate. That is, GuidExp generates code which is semantically equivalent to the target exploit.

Fig. 1b depicts the structure of our target exploit. The first portion of our target exploit consists of the *trigger slice*, which is the AS bytecode representation of the PoC. Note that while the trigger slice is able to drive the virtual machine in to a buggy state, entering to the buggy state is not sufficient for determining the severity of the bug or examining the way an attack would exploit the vulnerability. GuidExp aims to generate real exploit code that achieves the above by appending generated code to the bug-triggering code (the trigger



(a) Structure of a typical ROP attack **(b) Structure of our target exploit**

Figure 1: Exploit Structures

slice). The additional code required to build an exploit can vary from one attack to another, and is not necessarily small or simple.

Execution of the trigger slice causes vulnerable code segments in the AVM to be executed, but performs no further activity so as not to raise kernel panic. For a given vulnerability, GuidExp uses the same trigger slice as a prefix to an entire set of executables to be tested for potential exploit candidacy, thus it is crucial that the trigger slice avoids kernel panic, otherwise the generated executables would result in kernel panic causing our AEG process to fail.

The remaining part of the target exploit consists of a series of *exploit subgoals*—semantic goals for each step of the synthesized exploit; each exploit subgoal is used by GuidExp to synthesize code blocks that achieve that particular semantic goal. Together, the series of subgoals produce code that constitute the final exploit script. For example, a typical exploit subgoal in an ROP exploit (denoted by `'Corrupting a Buffer Space Implicitly'` in Fig. 1b) corrupts the size of a vulnerable buffer to read the memory beyond the buffer boundaries to gain access to `libc` libraries containing ROP gadgets [47].

Typical ROP attacks exploiting *use-after-free* (UAF) and *double-free* (DF) vulnerabilities in language virtual machines tend to follow a specific malicious activity pattern (a sequence of abstract logical steps). This established, well-rehearsed pattern allows for surreptitious penetration into the system, without being caught by standard operating system defenses. Here, first, the ROP attack script obtains one or more access privileges `-rwx-` for a system resource, such as reading privileges over `ELF` binaries. Then, by using these privileges, the ROP attack makes the next system resource, such as the `.plt` segment, which is located in `ELF` binaries available for itself. The ROP attack follows this pattern until being capable of completing its full malicious activity goal, such as invoking a system call. The fact that most exploits follow this typical pattern allows us to deconstruct exploit code into multiple exploit subgoals, whereby execution of each exploit subgoal sets the stage for the next exploit subgoal.

For example, in the exploit shown in Fig. 1b, the trigger slice, which exploits a UAF vulnerability, allows the ROP attack script to dereference a dangling pointer. The dangling pointer occurs after the UAF vulnerability is triggered. The dangling pointer points to the metadata of the freed buffer, so that the ROP attack can modify the metadata to corrupt the length of the buffer (see Section 5.2 for more details). The goal of the ROP attack is to change the `.length` property of the buffer *implicitly* with a large number, without explicitly calling the `.length` property. The implicit change in the `.length` property allows the ROP attack to gain access to memory that lies beyond the buffer boundaries, since the implicit change does not allow the AVM to allocate a large enough empty space for the new buffer size.

Corrupting the `.length` is our first exploit subgoal and denoted by `'Corrupting a Buffer Space Implicitly'` in Fig. 1b. Having the corrupted buffer allows the ROP attack to spray helper elements such as the payload to be executed into the heap, which is our second exploit subgoal and denoted by `Spraying Helper Elements` in Fig. 1b. The ROP attack follows this pattern until execution of its malicious payload, which is the last exploit subgoal, denoted by `'Building and Executing the ROP Chain'` in Fig. 1b.

2.3 Defining Exploit Subgoals, Search Spaces & Invariant

Since the semantics of "exploitability" is fluid, i.e., can change based on security engineers' expectations or security-sensitive assets, GuidExp provides flexibility in defining exploitability of target applications in various settings and environments. GuidExp allows defining exploitability as the successful completion of a series of exploit subgoals. For example, by providing exploit subgoals that are necessary to bypass ASLR, security engineers can obtain the exploit script, and then, they can see how the exploit code bypasses their ASLR implementation to fix their weaknesses. GuidExp expects such exploit subgoals to be defined by security experts who have a thorough knowledge of their target application since the success of GuidExp relies on defining the exploit subgoals accurately.

In order to synthesize code corresponding to each exploit subgoal, GuidExp takes as input a collection of exploit subgoals; each exploit subgoal consists of (1) a *search space* and (2) an *invariant*.

The search space consists of a set of *opcodes* and *parameters*. An opcode is the atomic portion of machine code instruction, in AS bytecode language, that specifies the operation to be performed. In AS language, opcodes take zero or more parameters to be used in the operation [9]. A parameter is either an index to a value stored in the constant pool of the executable or a constant to be pushed into the call stack directly. We expect that the security experts determines opcodes and parameters based on their experience. The experts should consider semantic meaning of every opcode and parameter and pick opcodes and parameters that can contribute to synthesizing the exploit subgoal.

An invariant is a test that decides whether the synthesized code semantically satisfies the corresponding exploit subgoal, and is written by the security expert in the form of an AS code snippet. GuidExp utilizes the invariant since it does not modify the implementation of the AVM or require recompiling the AVM to insert flags that alert when an error statement is reached.

Consider the simplified example of an exploit script containing an exploit subgoal of summing two known `integer` values. Assume, in this simplified example, the trigger slice for the exploit script creates these integers with the following code snippet:

```
1| function init(){
2|   var firstVariable = 6;
3|   var secondVariable = 12;
4| }
```

To achieve the exploit subgoal, GuidExp needs to append to the given PoC with the following:

```
1| var sum = firstVariable + secondVariable;
```

The line calculates the sum of given two `integer` variables, `firstVariable` and `secondVariable`. The same line consists of three smaller operations within: (1) assigning a value to a variable, since the resulting sum (`firstVariable + secondVariable`) is assigned to another variable (`sum`), (2) pushing the values to be summed onto the operand stack (since the AVM uses the operand stack to store temporary values), and (3) invoking the sum operator.

A security expert can therefore create the search space for this exploit subgoal by considering these subset of operations. The expert can choose these opcodes for the search space for the exploit subgoal: `getlocal`, `add`, and `setlocal`. The opcode `getlocal` pushes the value of local variables onto the operand stack, `add` is

the opcode that pops two values from the operand stack and pushes the result onto the operand stack, and `setlocal` pops the top value from the operand stack and assigns the value to a local variable. The parameters used with the opcodes should be the indices of the local variables. GuidExp is capable of calculating indices of exploit subgoal-relevant variables when their names are provided. If no variable name is provided, GuidExp calculates indices of all local and global variables and adds them to the current search space.

The invariant for this exploit subgoal tests whether the sum equals to a third known variable. A good invariant for the exploit subgoal could be:

```
1| return (sum == thirdVariable)
```

2.4 Constructing Exploit Script from Checkpoints

Once GuidExp synthesizes a code segment that satisfies the invariant for the current exploit subgoal, we declare that GuidExp achieved the exploit subgoal. Subsequently, GuidExp can move to the next subgoal. To do so, GuidExp first appends the synthesized code segment into the AS executable constructed so far. This combined executable is dubbed *checkpoint*. In this example, the checkpoint for the exploit subgoal consists of the PoC and the line that GuidExp synthesized in Section 2.3:

```
1| function init(){
2|   var firstVariable = 6; var secondVariable = 12;
3|   var sum = firstVariable + secondVariable;
4| }
```

Successfully acquiring a checkpoint enables GuidExp to be ready to aim for the next exploit subgoal; therefore, GuidExp can stitch the exploit script from checkpoint it synthesizes. When achieving one subgoal and findng a solution for the next one, GuidExp builds candidate slices on top of the solution found for the previous subgoal (i.e., new instruction permutations are appended to the checkpoint built from solving the previous subgoal). This guarantees that the solution for the new subgoal always satisfies the prior subgoals. GuidExp repeats this process until there is no more subgoal to achieve, then returns the last obtained checkpoint as an exploit script.

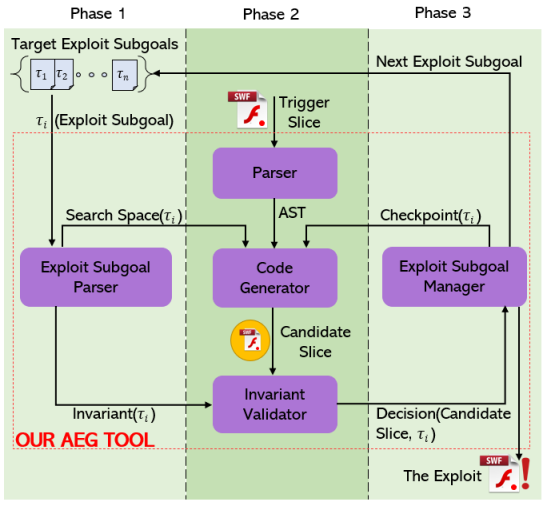


Figure 2: Overview of GuidExp

3 OVERVIEW OF GUIDEXP

Fig. 2 depicts an overview of GuidExp, which consists of three phases. GuidExp takes as input the full series of exploit subgoals, and at the end, produces the final exploit script. In the first phase, GuidExp reads an exploit subgoal (denoted by τ_i in Fig. 2) from the collection. Then, GuidExp parses the corresponding search space and the invariant (denoted by Search Space(τ_i) and Invariant(τ_i) in Fig. 2 respectively). The Exploit Subgoal Parser is responsible for taking the search space and the invariant from the exploit subgoal. Both the search space and the invariant are sent to different units to be used in the second phase.

In the second phase, GuidExp explores all possible execution paths that follow the execution of the trigger slice and checks whether the current exploit subgoal is achieved in any execution path. There are three main units in this phase: (i) the parser, which generates the abstract syntax tree (AST) from the trigger slice into Java structures; the AST becomes the input for the next main unit, (ii) the *Code Generator*, which analyzes the AST to locate the execution path in which the vulnerability is triggered. The Code Generator outputs executables that follow the execution path by appending a permutation of instructions given in the exploit subgoal to the trigger slice. The executables outputted by the Code Generator are input for the final main unit, (iii) the *Invariant Validator*, which dynamically monitors execution of the executables coming from the Code Generator to decide if the current exploit subgoal is achieved by any of them.

The Code Generator synthesizes distinct executable scripts, called *candidate slices* (denoted by Candidate Slice in Fig. 2), by appending distinct permutations of instructions given in the subgoal to the trigger slice at a time. Each executable script can explore a different execution path. However, at this point, GuidExp can generate an infinite number of candidate slices that follow the trigger slice. Therefore, along with the AST, the Code Generator receives as input the search space that consists a set of opcodes and parameters that can contribute to the task of satisfying the current exploit subgoal. GuidExp explores execution paths constructed with opcodes and parameters given in the search space. Thus, with having the search space, the Code Generator eliminates the execution paths that perform unrelated operations to the exploit. Candidate slices are appended to the trigger slice so that they trigger the vulnerability in the exact same way the trigger slice does.

Fig. 3 demonstrates how GuidExp explores execution paths. Here, q_i, red and gray nodes represent AVM program states. State q_0 is the initial state, and represents the initial settings of the AVM. The execution of the trigger slice transitions the program state to q_v, which occurs after the vulnerability is triggered. Then, GuidExp generates distinct candidate slices to explore new execution paths. The execution of every candidate slice results in a different program state, leading to one of three types of states:

(1) Red nodes represent program states that result in an error (e.g., type error, reference error, argument error) or perform an illegal call stack operation (e.g., pop when the call stack has zero elements). GuidExp does not append to the candidate slice whose executions

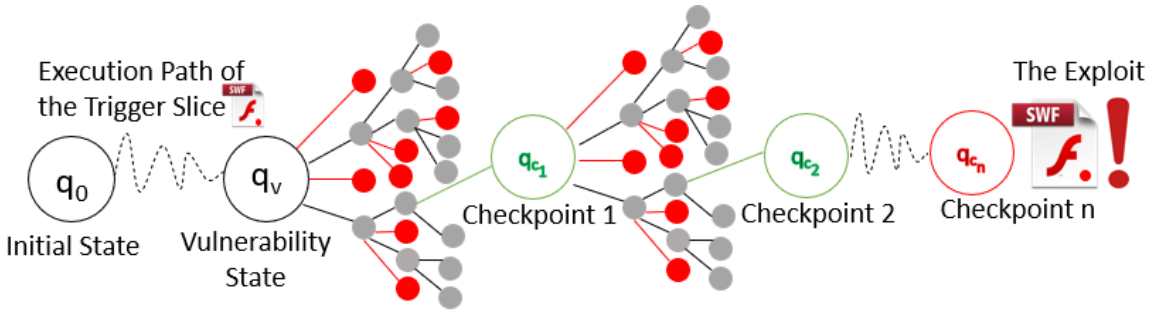


Figure 3: GuidExp Exploit Script Generation Process

terminate on a red node, since no matter what opcode is appended to the candidate slice, its execution raises the same error (see Section 6.4).
(2) Gray nodes represent program states that do not lead to a program error. Candidate slices that do not visit a red node are in both syntactically and semantically correct form, so they can be extended with more instructions to obtain new candidate slices. However, these candidate slices (that land on a gray node) cannot satisfy the current exploit subgoal. Thus, GuidExp needs to continue generating more candidate slices by appending new instructions to these candidate slices (of whose execution ends on a gray node).
(3) The candidate slice that satisfies the current exploit subgoal is denoted by a green node and "Checkpoint(τ_i)" in Fig. 3. When a checkpoint is synthesized, GuidExp stops generating further candidate slices for the current exploit subgoal, since it has already been satisfied. Then, GuidExp synthesizes new candidate slices to satisfy the next exploit subgoal. These candidate slices are generated by appending new instruction permutations to the checkpoint to follow the same execution path that satisfies the previous exploit subgoals. GuidExp, therefore, builds the exploit code (denoted by "The Exploit" in Fig. 3) by stitching the checkpoints after all of the given exploit subgoals are satisfied.

Generated candidate slices are sent to the Invariant Validator, which is the third main unit of the second phase and monitors runtime behaviors of candidate slices. As GuidExp does not modify the implementation of the AVM, it cannot make runtime observations. Therefore, GuidExp utilizes invariant to decide whether the corresponding exploit subgoal is satisfied. GuidExp inserts the invariant at the end of the execution of candidate slices to avoid altering their intended behaviors. We expect that the invariant would be given by security experts along with the search space as inputs for GuidExp. The result that the invariant generates (denoted by Decision (Candidate Slice, τ_i) in Fig. 2) is input for the *Exploit Subgoal Manager* which appraises the decision.

In the final phase, the execution result of candidate slices is evaluated by the Exploit Subgoal Manager. If the execution of a candidate slice results in an error, the AVM raises an error message. The error message indicates the type of the error with an error code [7]. GuidExp uses the error message to disqualify subsequently generated candidate slices based on the type of the error. If the result is a `false`, the result indicates that the candidate slice is executed without raising any error. However, the candidate slice does not achieve the corresponding target exploit subgoal. In this case, GuidExp discards the candidate slice and informs the Code Generator to synthesize a new candidate slice to be tested.

If the result is a `true`, the candidate slice (denoted by Checkpoint-(τ_i) in Fig. 2) achieves the corresponding target exploit slice. In this case, the Exploit Subgoal Manager stops the candidate slice generation process and informs the Exploit Subgoal Parser to parse the next target exploit subgoal. The Exploit Subgoal Parser reads the next search space and invariant. Simultaneously, the Exploit Subgoal Manager sends the candidate slice back to the Code Generator so that the Code Generator can use the candidate slice as the skeleton for the next exploit subgoal and this process keeps going until all target exploit subgoals are achieved.

4 PROTOTYPE IMPLEMENTATION

In Section 3, we present the overview of GuidExp 's main components. In this section, we provide additional implementation details for those components to provide the reader context for our experimental setup.

Code Generator. GuidExp's Code Generator differs from typical fuzzers in executable generation approach; while typical fuzzers randomly mutate given input seeds to generate new executables, the Code Generator performs *selective mutation* in which it intentionally alters code sequences in the trigger slice with a distinct instruction permutation given in the search space. Additionally, the Code Generator modifies the metadata of the generated executables

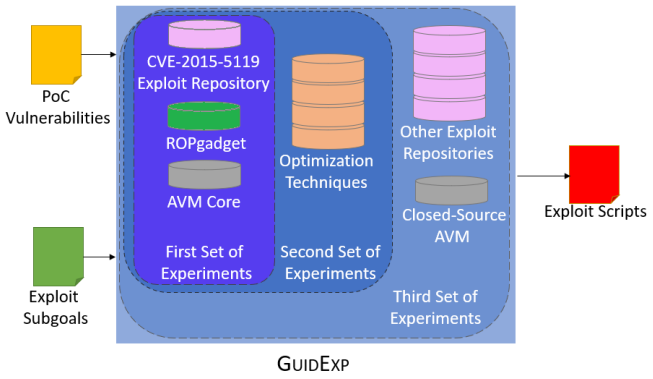


Figure 4: GuidExp Experimental Setup & Refinements

to ensure they do not violate the grammar rules enforced by the AVM, and can be executed without raising parsing errors.

Invariant Validator. As mentioned before, GuidExp 's Invariant Validator inserts the invariant at the end of the candidate slice to test whether the execution of the candidate slice achieves the current exploit subgoal. After inserting the invariant, it invokes a version of AVM by using Java's `ProcessBuilder` class [40], which is used to create operating system processes. To do that the Invariant Validator packs the AST of current candidate slice and creates the Flash executable from the AST. The Invariant Validator gives the location of the AVM implementation that runs the Flash scripts and the Flash script recently created. Invariant Validator allows a user to pick various versions of AVM.

Exploit Subgoal Manager. As mentioned above, the Exploit Subgoal Manager is responsible for evaluating the result of execution of candidate slices. Based on the result, it informs the Code Generator what to do next. Different versions of AVMs (mainly open-source versions vs closed-source versions) behave differently. Therefore, Exploit Subgoal Manager needs to recognize the differences and handle them appropriately. The open-source AVM versions can be executed in a terminal. This allows the Exploit Subgoal Manager to read the execution result from the terminal without running any I/O operation. However, the closed-source AVM versions are distributed as parts of Flash Players. They do not return any information to the terminal but use pop-ups to inform users in case an error is raised. Therefore, the Exploit Subgoal Manager cannot know whether an error is raised during the execution of a candidate slice. Additionally, the closed-source AVM versions keep logs of the execution of Flash scripts in the file system. The Exploit Subgoal Manager is required to use I/O operations to reach these logs to learn about the execution results of Flash scripts. These I/O operations significantly slowdown the exploit generation task, when GuidExp is running with a closed-source AVM.

5 INITIAL EXPERIMENTAL SETUP

The main goal of the initial experiment setup was to allow quick evaluation and validation of various design ideas without paying the expense of a full experiment. In order to make the iteration as quick as possible, we hand-crafted a single test achieving the balance between coverage and the simplicity.

In the initial experiments we collected three pieces of data: (1) the number of candidate slices GuidExp generates, (2) the number of candidate slices executed successfully, (3) and the time GuidExp needs to take to achieve one exploit subgoal. To do that we borrowed three artifacts from the community and in the rest of the section, we introduce each of them individually and how we used them.

5.1 ROP Gadget Sequence

We want GuidExp to synthesize the exploit script building and executing an ROP sequence. At this point, there are many different ways of building ROP sequences each of which performs different types of ROP attack. We decided to build the ROP sequence ROPgadget [29] builds because it is not too complicated, and its execution can be observed without debugging the AVM. ROPgadget builds an ROP sequence from gadgets it locates and that can be accessed during the execution of given binary. We ran ROPgadget in our system and extracted the ROP sequence it built. We used the ROP sequence to construct the exploit subgoal that allows GuidExp to synthesize an AS code block that executes the same ROP sequence, then GuidExp appends the AS code block to the exploit script it generates.

GuidExp aims to synthesize an exploit script that performs an ROP attack. ROP attacks can perform different types of malicious activities based on the sequence of gadgets (also known as the *ROP chain*) they execute, e.g., producing a shell, running arbitrary code or invoking a system call. Therefore, an ROP attack needs to build the correct gadget sequence to achieve its malicious intention. GuidExp builds the ROP chain that executes `'int 0x80'`, which is used to invoke system calls. GuidExp builds and executes the ROP chain in the final exploit subgoal, `'Building and Executing the ROP Chain'`. The ROP chain consists of 38 lines of codes and contains ten distinct gadgets. GuidExp builds the chain by itself after locating these ten gadgets. To locate a gadget, GuidExp needs to synthesize a function which scans `libc` libraries and returns the address of the given gadget. After locating the first gadget, GuidExp invokes the same function definition with different gadget to locate all required gadgets.

5.2 Initial Vulnerability & PoC Exploit Script

For our initial experimental setup, we chose CVE-2015-5119 [39] as our target vulnerability for several reasons. First, the vulnerability is a real-world vulnerability and was exploited frequently by exploit kits that demonstrates the impact of the vulnerability. In fact, CVE-2015-5119 was one of the Kaspersky's Devil's Dozen Flash vulnerabilities that gained immense popularity among criminals and was added to numerous exploit kits in 2015 [30]. Second, the vulnerability is simple enough that allows us to comprehend all aspects of exploiting vulnerable AVM versions. Third, the vulnerability is not too simple and creates many challenges that allows us to be familiar with issues that we can encounter when we work different vulnerabilities.

```
1  public class malClass extends Sprite {
2    public function malClass() {
3      var b1 = new ByteArray();
4      b1.length = 0x200;
5      var mal = new hClass(b1);
6      b1[0] = mal;
7    }
8
9    public class hClass {
10     private var b2 = 0;
11     public function hClass(var b3) {
12       b2 = b3;
13     }
14     public function valueOf() {
15       b2.length = 0x400;
16       return 0x40;
17     }
18   }
19 }
```

Listing 1: PoC Exploit for CVE-2015-5119

The target vulnerability resides in the implementation of AVM versions up to 18.0.0.194 for Windows, OS X, and Linux [10]. The vulnerability happens due to a lack of a control mechanism of side

effects of implicit function calls, e.g., invoking `valueOf()` to get the value of an instance while assigning it to another instance.

Listing 1 shows how the vulnerability is triggered. The class `malClass`, which invokes malicious `valueOf()`, creates a `ByteArray` instance, `b1`, and sets its length as 0x200 in Line 3, 4. A `ByteArray` instance is a packed array of bytes that has methods and properties to optimize working with binary data. Line 5 creates an instance which belongs to `hClass`, with `b1` as its attribute and assigns it to the index 0 of `b1`. In Line 14, the `valueOf()` function is overridden to free `b2` attribute of `hClass` instances by altering its length. The AVM memory management system prefers first to deallocate the object, and then to reallocate it to a bigger memory chunk in case it needs a bigger memory space. Therefore, the assignment happens in Line 6, leading to freeing `b1`, and reallocating it to a bigger memory chunk, since the length of `b2` (0x400) is now larger than it was (0x200). However, the AVM does not check this side effect, so the index 0 of `b1` still references the freed memory chunk, allowing writing the return value (0x40) to the freed memory chunk.

Since GuidExp has to be guided with exploit subgoals to achieve exploit generation task, in the initial experimental setup, we prepared the exploit subgoals by analyzing a well-studied exploit script [44] provided by the cybersecurity company Rapid7 [43]. This exploit script exploits the CVE-2015-5119 vulnerability and runs an ROP attack that allows attacker to perform an arbitrary code execution. We observed the execution of the exploit script and extracted the methodology of setting the scene for running the ROP chain after triggering the vulnerability. In our experiments we prepared the exploit subgoals that follow the methodology we extracted from the exploit script we analyzed. Our exploit subgoals leads GuidExp to synthesize a new exploit script that triggers the vulnerability we introduced in this section, and then mimics the ROP attack explained in Section 5.1. We did not build the ROP chain given in the exploit script but preferred to execute the ROP chain ROPgadget constructs (see Section 5.1).

5.3 Vulnerable AVM Version

Although there are numerous AVM versions been published, there is only one open-source version and the other versions are kept as proprietary by the owner company. While we used the open-source AVM version in our initial experiments, we exploit a closed-source AVM version later, and we discuss these experiments in Section 7.2.

In our initial experiments, we used the *AVM Core* version as our vulnerable VM, which is the only open-source AVM version available in GitHub [3], commit '65a0592'. The AVM Core includes only the core functionalities such as parsing and running Flash scripts, but no external libraries, such as libraries required to run the GUI.

Here, GuidExp explores a new execution path by executing a distinct candidate slice it synthesized. In our initial experiments, we used the AVM Core to execute candidate slices GuidExp synthesized and evaluated execution logs the AVM Core generated to learn about the execution results of candidate slices.

We chose the AVM Core because it runs significantly faster than the closed-source versions since it does not need to run the external libraries every time it starts-up. Additionally, to our knowledge, the AVM Core contains only one vulnerability, which is the vulnerability we introduced in Section 5.2; this allows us to quickly iterate over exploiting our target vulnerability. However, as discussed in Section 7, we finally conducted several experiments with the closed-source versions.

5.4 Results

In our initial experiments, we aimed to have a working system which is fast and lean. We collected three data pieces, including the time GuidExp needs to take to achieve the first exploit subgoal. As shown in Fig. 4, in the first set of experiments we did not adopt any optimization techniques. Unfortunately, the initial experiment was timed out after we let GuidExp run more than a day. However, we got three important results from the initial experiment. First, a significant majority of candidate slices raised an error without completing their execution, which allowed us to identify sequences of instructions that cause these errors. Therefore, we disqualified candidate slices that include one of these error-raising instruction sequence before we execute them (see Section 6.2 and 6.4 for more details).

Second, with having the first pseudo-exploit subgoal, we realized that we can deconstruct the exploit script into many smaller exploit subgoals and GuidExp can focus on synthesizing exploit scripts achieving for each exploit subgoal individually. This significantly reduced the number of candidate slices GuidExp needed to synthesize to achieve all exploit subgoals (see Section 6.1).

Third, to deconstruct the exploit, at first, we manually chopped the exploit script and analyze instructions in each piece. Our analyses show us that the number of different instructions used to achieve each subgoal was significantly smaller than the number of different instructions used in the entire exploit. This implied that a seasoned security expert who has thorough knowledge on ActionScript instructions and exploits can restrict the instructions to be used in synthesizing each exploit subgoal individually (see Section 6.1).

Although the initial experiment was timed out and we did not allocate enough time for finishing it (approximately 3 years with our rig), three bottom lines of the initial experiment allow us to derive the optimization techniques that we introduce in Section 6.

6 DERIVING OPTIMIZATION TECHNIQUES

Finding the correct permutation of instructions given in exploit subgoals requires testing all possible permutations in the worst case. According to the exploit script [44] we analyze, the first exploit subgoal contains nine opcodes and six parameters, and the bytecode sequence satisfying the exploit subgoal consists of twelve instructions. Hence, GuidExp must generate and run 54^{12} candidate slices in the worst case.

To deal with such a search space blow up, we have developed four optimization techniques, guided by data collected from the initial experiments. In this section, we introduce these four optimization techniques along with the observations that lead us to the techniques.

6.1 Deconstructing an Exploit into Subgoals

The very first version of GuidExp didn't have a notion of exploit subgoals. Instead, we attempted to use GuidExp to construct the entire exploit code at once, and the attempt was not successful; the GuidExp just kept running for days without getting any result until we killed the process. Facing the challenge, we first tried to synthesize a small prefix of the full exploit. We soon realized it is possible to construct the exploit in a piecemeal. This observation lead us to the *exploit deconstruction*.

With exploit deconstruction, as mentioned in Section 2.3 and demonstrated in Fig. 1b, the task of generating a full exploit script is split into a task of sequentially generating many smaller exploit subgoals. In this approach, GuidExp iteratively finds a shortest candidate slice that reaches the checkpoint for each exploit subgoal and stitches them to build a full exploit script. When GuidExp synthesizes a candidate slice that reaches a checkpoint, GuidExp prunes all other execution paths that cannot reach the checkpoint, or those that need a longer path to reach the checkpoint.

Consider the example in Fig. 3. The example demonstrates how exploit deconstruction prunes execution paths that GuidExp needs to explore. After reaching the Checkpoint(τ_1), GuidExp focuses on synthesizing the candidate slice that reaches Checkpoint(τ_2) through Checkpoint(τ_1), although it is possible that there are execution paths that do not visit Checkpoint(τ_1) but do reach Checkpoint(τ_2). However, the number of execution paths that GuidExp needs to explore increases exponentially in each level as GuidExp appends the permutations of instructions given in the current search space to the trigger slices. Thus, with having exploit deconstruction, our experiments show that we disqualify the vast majority of execution paths.

6.2 Operand Stack Verification

Exploit deconstruction made GuidExp start to emit some results, but the performance was not satisfactory: GuidExp still took about a half day to find a solution slice for a single subgoal.

We analyzed the list of candidate slices discarded by GuidExp, and observed that 98% of discarded candidate slices were rejected by AVM because of *operand stack errors*. The observation lead us to implement built-in operand stack verification. The main benefit of a built-in verification is that it can run before running AVM, allowing us to disqualifying erroneous candidate slices without paying the AVM execution overhead.

AVM is a stack machine, where the operand stack holds operands for the instructions and stores results of operations. If a wrong combination of operand stack operations are performed, AVM detects the problem and raise one of the following two errors: (1) *stack underflow*, which occurs when an instruction tries to pop elements from the operand stack while the operand stack holds no element, (2) *stack overflow*, which occurs when a function returns before popping all elements it pushed onto the operand stack.

In *operand stack verification*, GuidExp simulates the operand stack for the candidate slice it generates to decide whether the candidate slice causes an operand stack violation *before* sending the candidate slice to the Invariant Validator. If a candidate slice causes the stack underflow error, GuidExp marks the instruction permutation that the candidate slice contains as `ill-prefix` and discards the candidate slice. GuidExp also eliminates the subsequently generated candidate slices which contain an `ill-prefix` instruction permutation since instruction sequences are prefix-closed, and will raise the same error. If a candidate slice causes the stack overflow error, GuidExp eliminates the candidate slice but does not mark the instruction permutation it contains as `ill-prefix`, because candidate slices that cause stack overflow error might be followed by instruction sequences that consume remnant elements in the operand stack.

To further optimize operand stack verification, GuidExp remembers all observed `ill-prefixes`. Given a new candidate slice, GuidExp checks whether the slice contains any previously observed `ill-prefix`. If the slice contain a previously discovered `ill-prefix`, GuidExp rejects it immediately. If the slice does not contain any known `ill-prefix`, but the operand stack verification gives an error, GuidExp adds the slice to the `ill-prefix` set and rejects the slice. Otherwise, GuidExp continues to execute the slice in AVM.

GuidExp uses an *m-ary trie*, demonstrated in Fig. 5, to concisely represent an `ill-prefix` set. Here, `m` is the number of instructions in the current search space and `n` is the maximum depth we allow GuidExp to explore the execution-path space. In each search space, the instructions are given a unique integer identifier. So, the trie stores these unique identifiers as the value of the nodes in the `ill-prefix` trie. We prefer a trie structure because it is memory efficient, and it provides an efficient search operation. We initially stored the discovered `ill-prefixes` in a `nested linkedlist`, which has the time complexity of $O(m^n)$ for searching/inserting and the space complexity of $\Theta(m^n)$. That caused GuidExp to spend significant time in searching for a matching `ill-prefix`, and to consume several GB of memories remembering `ill-prefix`. Thus, the idea of remembering observed `ill-prefixes` lost its meaning. On the contrary, `m-ary trie`'s search operation has $\Theta(n)$ time complexity. As a result, GuidExp spends significantly less time in searching/inserting for an `ill-prefix`, that allows us to save significant time. `Trie` also has a smaller memory consumption: $O(m^{n-1})$.

6.3 Instruction Tiling

Instructions in AS bytecode typically need to be used in particular sequences, together, to represent semantically meaningful activities. E.g., the opcode `"add"`, which pops two values from the top of the operand stack and then pushes the result back to the operand stack, requires that these two values be pushed onto the operand stack

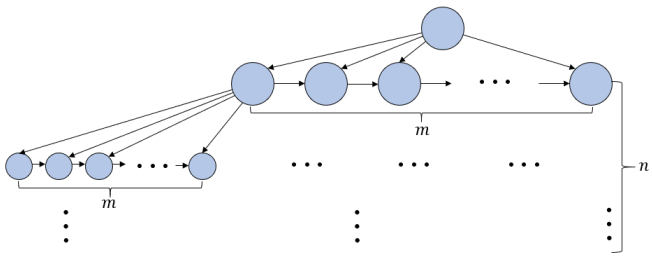


Figure 5: The structure of the `ill-prefix` trie

previously. Therefore, the opcode "add" and the opcode "push" are commonly used together to perform the summation.

Our third optimization technique, *instruction tiling*, uses such relationships between instructions, to create instruction chains that can perform meaningful activities such as calling a variable, coercing a type of variable, or calling a property of an object. We refer to such an instruction chain as a *tile*. GuidExp generates candidate slices adding or replacing a tile instead of an instruction. Thus, the number of candidate slices that GuidExp synthesizes decreases dramatically as the number of permutations of tiles is significantly smaller than the number of permutations of instructions. GuidExp expects security experts to specify tiles using their expertise on ActionScript semantics.

6.4 Feedback from the AVM

The Code Generator sends candidate slices that do not violate the operand stack to the Invariant Validator to be executed in the AVM in Phase 2 in Figure 2. However, the AVM can raise different types of run-time errors during the execution of candidate slices that GuidExp cannot detect before their execution. The AVM raises these errors when candidate slices perform an illegal operation, such as reading outside of an array boundaries, or if the AVM cannot keep running because of resources restrictions. For example, if a candidate slice contains an infinite loop, the AVM will raise the `out-of-memory` error. The Code Generator marks the instruction permutation that the error-raising candidate slice contains as `ill-prefix`, and discards the candidate slice. The Code Generator also inserts `ill-prefixes` to the `ill-prefix` trie the same way it handles `ill-prefixes` that violate the operand stack.

The most common types of error messages that GuidExp receives are: `TypeError` [8], `ArgumentError` [2], `ReferenceError` [6], `RangeError` [5], stack underflow, and stack overflow errors [9].

TypeError. A `TypeError` is thrown when the actual type of an operand is different from the expected type. In addition, this exception is raised when a value is assigned to a variable and cannot be coerced to the variable's type or the `super` keyword is used illegally. Candidate slices that raise a `TypeError` are eliminated after they are executed in the AVM. However, the other candidate slices that append to these candidate slices are disqualified even without being generated by GuidExp. Although the type of operands can be coerced explicitly, GuidExp allows type coercion only before the operand is called [8].

ArgumentError. An `ArgumentError` occurs when the arguments supplied in a function do not match the arguments defined for that function. This error is raised, for example, when a function is called with the wrong number of arguments, an argument of the incorrect type, or an invalid argument. When an `ArgumentError` is raised, GuidExp verifies the candidate slice that raises the error because the error might occur because of a wrong number of arguments. GuidExp changes the bytecode that declares the number of arguments with the number of elements in the operand stack when the function is called. After the candidate slice is verified, it is executed one more time to see if the error persists. If yes, the candidate slice and the other candidate slices that append to the candidate slice are disqualified as the error occurs because of the incorrect type or invalid arguments [2].

ReferenceError. A `ReferenceError` exception is thrown when a reference to an undefined `property` is attempted. Candidate slices that GuidExp generates raise this error when a `property` is called by an object that does not define the `property` it calls. The other candidate slices that follow the candidate slice that raises a `ReferenceError` must remove the reference of the undefined `property`. However, GuidExp does not allow such an operation because pushing an element onto the operand stack and then popping it without making use of it can create infinite loops [6].

RangeError. A `RangeError` occurs when an invalid index is provided to an array type buffer. An index is invalid if it is less than zero, or it points beyond the array boundaries, or it is not an `integer`. When a candidate slice causes the AVM to raise a `RangeError`, after discarding the candidate slice, GuidExp disqualifies the other candidate slices that append to the candidate slice, based on the type of the index [5].

7 EXPERIMENT REFINEMENTS

In this section, we present our final experiments with refinements. With each refinement, we improve the performance of GuidExp, and increase the coverage.

All experiments were conducted on a virtual machine with a 3.4 GHz Intel Core i7 processor with 8 GB RAM. We used VMware Workstation 15 to emulate the virtual machine with Ubuntu 16.04 LTS. PoC scripts were created using Adobe Flex SDK 4.6 [4], `mxmlc`, and Mozilla Tamarin Project AS Compiler, `asc.jar` [38]. GuidExp was written in Java with NetBeans IDE 8.0.2 JDK v.1.8.

7.1 Refinement #1: Initial PoC

In the second set of experiments, we adopted the optimization techniques we introduced in Section 6. We repeated the first set of experiments with the optimization techniques and we used the same PoC vulnerability to exploit (see Section 5.2 for more details).

After refining GuidExp with the optimization techniques, we conducted the second set of experiments to see whether synthesizing the exploit script was feasible for GuidExp.

GuidExp utilized the AVM Core, the open-source AVM implementation, to execute candidate slices GuidExp generated for our example vulnerability. Table 1 demonstrates our experimental results with the AVM Core for our example vulnerability. We give the number of generated candidate slices and executed candidate slices during synthesizing each exploit subgoal. GuidExp output the exploit script in just under 15 minutes.

7.2 Refinement #2: Closed-Source AVM Versions

Exploit Databases. To conduct the third set of experiments, we used PoC exploit scripts for known AVM vulnerabilities, selected from exploit databases such as exploit-db [21] and Google Project Zero [42]. We selected CVE-2013-0634, CVE-2014-0502, -0515, -0556, CVE-2015-0313, -0359, -3090, -3105, -5122 since they are well-studied and are included in commonly used exploit kits.

Exploit Subgoal	Number of Generated Candidate Slices	Number of Executed Candidate Slices		Percentage of Executed Candidate Slices		Synthesizing Time (s)	
		open-source	closed-source	open-source	closed-source	open-source	closed-source
Corrupting a Buffer Space Implicitly	2,396,744	12,229	29,167	0.51	1.21	9.35	605.58
Spraying Helper Elements	19,173,952	73,997	210,225	0.38	1.09	55.90	3,895.64
Locating Sprayed Elements	37,448	357	769	0.95	2.05	1.72	12.76
Disclosing the Offset of the Located Elements	55,345,757	282,392	508,339	0.51	0.91	138.26	6,845.86
Corrupting the Disclosed Buffer	4,793,488	21,591	41,342	0.45	0.86	17.03	963.86
Locating ELF Object Files	19,173,952	81,545	201,852	0.42	1.05	57.12	3,364.89
Locating `libc` Libraries	55,345,757	278,385	459,336	0.50	0.82	138.05	6,276.25
Locating Executable Segment	76,695,808	379,587	706,031	0.49	0.92	199.78	9,546.07
Locating Gadgets and Building the ROP Chain	435,848,049	1,648,451	2,954,400	0.37	0.67	240.92	11,512.47
					Total Time (with the open-source AVM implementation):		858.13 (14m 18.13s)
					Total Time (with the closed-source AVM implementation):		43,023.38 (11h 57m 03.38s)

Table 1: Exploit generation for CVE-2015-5119 with open-source core implementation of the AVM and closed-source Flash Player

We used the PoC exploit scripts for the vulnerabilities to prepare accurate exploit subgoals. We aimed to demonstrate that GuidExp can generate new exploit scripts for a given vulnerability using a PoC exploit script and defined exploit subgoals.

Closed-Source AVM Versions. To our knowledge, the AVM Core contains only one vulnerability, which is the vulnerability we used in our first and second sets of experiments. Therefore, we could not conduct further experiments to demonstrate GuidExp is not restricted to synthesizing the exploit script for only one vulnerability.

To overcome this challenge, we exploited vulnerabilities in the closed-source AVM versions, also known as Flash Player. Flash Player v11.2.202.262 contains all of these vulnerabilities, therefore, we could conduct our experiments without changing the setup. Running a closed-source AVM brought two interesting changes.

First, exploit generation process took around 45 times longer as compared to our second set of experiments. Table 1 also shows our experimental results with the closed-source Flash Player, v11.2.202.262, for our example vulnerability. The slowdown is due to the starting/closing overhead of the Flash Player. Note that there is no easy way to just start/stop the AVM included in the closed Flash Player. To run an AS executable on the closed-source AVM, we have to start/stop the full Flash Player every time GuidExp generates a candidate slice. Specifically, starting and closing a Flash Player takes 85ms on average, equivalent to ∼ 89% of the time required to test one candidate slice, producing the slice takes ∼ 1%, executing the slice takes ∼ 6%, reading the result takes ∼ 4%. Con the contrary, the open-source version has a smaller initialization overhead. The initialization overhead only takes ∼ 11% of the experiment for open-source AVM. Note that this performance difference is due to the characteristics of AVM, and it is not an issue of GuidExp. In fact, any mutation testing tool and fuzz testing tool targeting a closed-source AVM will have to deal with the initialization/termination overhead.

Second, since the error messages that the player outputs are shown to the users in pop-ups, we cannot leverage the feedback coming from the player. Thus, the number of candidate slices to be searched is higher with the player. Again, this is a characteristic of the closed-source AVM version. GuidExp may easily fetch the error messages raised by closed-source versions of other language virtual machines.

Results. Table 2 shows our experimental results with eleven AVM vulnerabilities we selected. In these experiments, GuidExp executes candidate slices with the closed-source player. According to our experiments, GuidExp can generate an exploit script for a vulnerability in less than 14 hours.

Table 2: Exploit generation results for selected vulnerabilities

Selected Vulnerabilities	Synthesizing Time	Flash Player Version
CVE-2015-5119	**11h 57m 03.38s**	**v11.2.202.262**
CVE-2013-0634	12h 09m 14.50s	v11.2.202.262
CVE-2014-0502	12h 54m 15.19s	v11.2.202.262
CVE-2014-0515	12h 51m 26.67s	v11.2.202.262
CVE-2014-0556	12h 08m 35.29s	v11.2.202.262
CVE-2015-0311	11h 56m 19.10s	v11.2.202.262
CVE-2015-0313	12h 20m 47.98s	v11.2.202.442
CVE-2015-0359	11h 05m 05.61s	v11.2.202.262
CVE-2015-3090	12h 01m 33.16s	v11.2.202.262
CVE-2015-3105	13h 25m 46.80s	v11.2.202.262
CVE-2015-5122	12h 07m 02.59s	v11.2.202.262

7.3 Refinement #3: Initial PoC with a Larger Search Space

To demonstrate that GuidExp can tolerate some level of inaccuracy in defining a search space, we performed additional experiments. This is important because it is unrealistic to expect a human security expert to always provide the tightest search space without any mistake. In this experiments, we run GuidExp with 1.25, 1.5, and 2 times larger search spaces than the optimal search space (the most accurate).

Table 3 shows exploit script synthesizing time of GuidExp with larger spaces for CVE-2015-5119 with the AVM Core. Since GuidExp generates a candidate slice for every permutation of instructions and parameters given in search spaces, the performance of GuidExp is affected by combinatorial rate with having unnecessary instructions and parameters in search spaces.

7.4 Discussions

It turned out we were lucky that we started with a vulnerability in an open-source VM because with a closed-source VM, our initial development could be infeasible. Therefore, we learned that

Table 3: Subgoal synthesizing time(s) with different size of search spaces

Exploit Subgoal	Base Search	1.25x Size of Search Space	1.5x Size of Search Space	2x Size of Search Space
Corrupting a Buffer Space Implicitly	**9.35**	44.58	161.75	1,184.6
Spraying Helper Elements	**55.90**	310.05	1184.51	10,120.85
Locating Sprayed Elements	**1.72**	6.70	26.82	108.45
Disclosing the Offset of the Located Elements	**138.26**	827.14	3543.43	35,417,77
Corrupting the Disclosed Buffer	**17.03**	62.18	270.18	1104.61
Locating ELF Object Files	**57.12**	308.14	1,204.95	10,348.54
Locating libc Libraries	**138.05**	820.98	3,607.04	34,178.59
Locating Executable Segment	**199.78**	1,230.68	5,319.27	52.980.04
Locating Gadgets and Building the ROP Chain	**240.92**	1,679.20	7,252.01	67,518.73
Total:	**858.13 (14m 18.13s)**	5,289.65 (1h 28m 9.65s)	22,569.96 (6h 16m 09.96s)	212,962.18 (59h 10m 03.38s)

finding a good "golden" example lets us to focus on it to accelerate development. This allows us to iterate extremely fast, and lead us to understand all the small details of algorithms and examples. However, our initial development could be biased to a wrong direction, if our "golden example" were not providing enough coverage and not raising issues that we can encounter with different vulnerabilities.

Additionally, generalizing a number obtained from one configuration to another configuration can be challenging. To resolve this challenge, we need to conduct another experiment to explain the difference between two configurations and the reasons why we could not use the numbers from the previous experiments.

8 CONCLUSION AND FUTURE WORK

In our recent work, we proposed the first semi-automatic exploit generation tool, GuidExp, for AVM vulnerabilities. We demonstrated that GuidExp successfully exploited eleven different AVM vulnerabilities in open- and closed-source AVM implementations. In this work, we present additional details on our development cycles, how we improved the performance number of GuidExp, after each iteration, and introduce the artifacts that we borrow from the community.

In future work, we plan to conduct a user study where we justify and measure the human intervention cost. Our user study will focus on answering questions such as "How much expertise is needed to set up exploit subgoals?", "To what extent the proposed framework can help craft exploits and determine exploitability?", or "Since we have exploit patterns to follow, what the role that human plays in the procedure?".

ACKNOWLEDGMENTS

This research was supported by NSF CRII award #1566321; we would like to thank Dr. Koushik Sen for providing help.

REFERENCES

[1] Anno Accademico. Static detection and automatic exploitation of intent message vulnerabilities in android applications. Master's thesis, Politecnico Di Milano, 2013.
[2] Adobe Inc. ArgumentError - AS3. https://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/ArgumentError.html. Accessed: 2021-02-11.
[3] Adobe, Inc. avmplus. https://github.com/adobe/avmplus.
[4] Adobe, Inc. Download Adobe Flex SDK. https://www.adobe.com/devnet/flex/flex-sdk-download.html.
[5] Adobe Inc. RangeError - AS3. https://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/RangeError.html. Accessed: 2021-02-11.
[6] Adobe Inc. ReferenceError - AS3. https://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/ReferenceError.html. Accessed: 2021-02-11.
[7] Adobe, Inc. Run-Time Errors. https://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/runtimeErrors.html.
[8] Adobe Inc. TypeError - AS3. https://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/TypeError.html. Accessed: 2021-02-11.
[9] Adobe, Inc. ActionScript Virtual Machine 2 (AVM2) Overview. https://www.adobe.com/content/dam/acom/en/devnet/pdf/avm2overview.pdf, May 2007.
[10] Adobe, Inc. Adobe security bulletin. http://tinyurl.com/ofdwo9c, July 2015. Accessed" 2016-12-03.
[11] Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and V.N. Venkatakrishnan. Chainsaw: Chained automated workflow-based exploit generation. In *Proceedings of the 23th ACM Conference on Computer and Communications Security (CCS)*, October 2016.
[12] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. Aeg: Automatic exploit generation. In *Proceedings of The Network and Distributed System Security Symposium (NDSS)*, February 2011.
[13] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. Finding software vulnerabilities by smart fuzzing. In *Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2011.
[14] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP'14)*, May 2014.
[15] Konstantin Böttinger and Claudia Eckert. Deepfuzz: Triggering vulnerabilities deeply hidden in binaries. *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 9721:25–34, 2016.
[16] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, pages 27–38. ACM, 2008.
[17] Dan Caselden, Alex Bazhanyuk, Mathias Payer, Stephen McCamant, and Dawn Song. HI-CFG: Construction by Binary Analysis and Application to Attack Polymorphism. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS)*, pages 164–181, 2013.
[18] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (ACM CCS)*, pages 952–963, 2015.
[19] Jared D. DeMott, Richard J. Enbody, and WIlliam F. Punch. Towards an automatic exploit pipeline. In *Proceedings of the 6th International Conference for Internet Technology and Secured Transactions (ICITST)*, December 2011.
[20] Pavel Dovgalyuk, Denis Dmitriev, and Vladimir Makarov. Don't panic: reverse debugging of kernel drivers. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, 2015.
[21] Exploit-DB. Exploit database - exploits for penetration testers, researchers, and ethical hackers. https://www.exploit-db.com/. Accessed: 2021-02-11.
[22] Joshua Garcia, Mahmoud Hammad, Negar Ghorbani, and Sam Malek. Automatic generation of inter-component communication exploits for android applications. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, September 2017.
[23] Sean Heelan. Automatic generation of control flow hijacking exploits for software vulnerabilities. Master's thesis, University of Oxford, 2011.
[24] Sean Heelan, Tom Melham, and Daniel Kroening. Automatic heap layout manipulation for exploitation. In *Proceedings of the 27th USENIX Security Symposium (USENIX SS)*, August 2018.
[25] Hu Hong, Chua Zheng Leong, Adrian Sendroiu, Saxena Prateek, and Liang Zhenkai. Automatic generation of data-oriented exploits. In *Proceedings of the 24th USENIX Conference on Security Symposium (USENIX SS)*, pages 177–192, August 2015.
[26] Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang, Chung-Wei Lai, Han-Lin Lu, and Wai-Meng Leong. Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. In *2012 IEEE Sixth International Conference on Software Security and Reliability*, pages 78–87. IEEE, 2012.

[27] Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang, Han-Lin Lu, and Chung-Wei Lai. Software crash analysis for automatic exploit generation on binary programs. *IEEE Transactions on Reliability*, 63(1):270–289, 2014.

[28] Karthick Jayaraman, David Harvison, and Adam Kiezun Vijay Ganesh. jFuzz: A concolic whitebox fuzzer for Java. In *Proceedings of the First NASA Formal Methods Symposium*, 2009.

[29] JonathanSalwan. Ropgadget. https://github.com/JonathanSalwan/ROPgadget.

[30] Kaspersky security bulletin 2015. The overall statistics for 2015. http://tinyurl.com/zgkkdbj.

[31] Cha Sang Kil, Avgerinos Thanassis, Rebert Alexandre, and Brumley David. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP'12)*, pages 380–394, 2012.

[32] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, july 1976.

[33] Andrea Lanzi, Lorenzo Martignoni, Mattia Monga, and Roberto Paleari. A Smart Fuzzer for x86 Executables. In *Proceedings of the 29th International Conference on Software Engineering Workshops (ICSEW)*, 2007.

[34] Lannan Luo, Qiang Zeng, Chen Cao, Kai Chen, Jian Liu, Limin Liu, Neng Gao, Min Yang, Xinyu Xing, and Peng Liu. System service call-oriented symbolic execution of android framework with applications to vulnerability discovery and exploit generation. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, June 2017.

[35] Lannan Luo, Qiang Zeng, Chen Cao, Kai Chen, Jian Liu, Limin Liu, Neng Gao Min Yang, Xinyu Xing, and Peng Liu. Context-aware system service call-oriented symbolic execution of android framework with application to exploit generation. *CoRR*, 2016.

[36] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[37] MITRE, Inc. CVE details - The ultimate security vulnerability datasource. https://www.cvedetails.com/vulnerability-list.php?vendor_id=53&product_id=6761&version_id=&page=1.

[38] Mozilla.org. Tamarin project. https://www-archive.mozilla.org/projects/tamarin/.

[39] National Institute of Standards and Technology. CVE-2015-5119. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-5119, 2015.

[40] Oracle. Processbuilder (java platform se 7). https://docs.oracle.com/javase/7/docs/api/java/lang/ProcessBuilder.html. Accessed: 2021-02-11.

[41] V. A. Padaryan, V. V. Kaushan, and A. N. Fedotov. Automated exploit generation for stack buffer overflow vulnerabilities. *Programming and Computer Software*, 41, 2015.

[42] project-zero. Monorail - project-zero - project zero - monorail. https://bugs.chromium.org/p/project-zero/issues/list. Accessed: 2021-02-11.

[43] Rapid7. Cybersecurity and compliance solutions and services | rapid7. https://www.rapid7.com/. Accessed: 2021-02-11.

[44] Rapid7. metasploit-framework/cve-2015-5119. https://github.com/rapid7/metasploit-framework/tree/master/external/source/exploits/CVE-2015-5119. Accessed: 2021-02-11.

[45] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, volume 17, pages 1–14, 2017.

[46] Dusan Repel, Johannes Kinder, and Lorenzo Cavallero. Modular synthesis of heap exploits. In *Proceedings of the 12th ACM SIGSAC Workshop on Programming Languages and Analysis for Security (PLAS)*, 2017.

[47] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[48] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security (CCS)*, pages 552–561, 2007.

[49] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. "sok:(state of) the art of war: Offensive techniques in binary analysis". In *Proceedings of the 37th IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016.

[50] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP'13)*, May 2013.

[51] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of 23rd Annual Network and Distributed System Security Symposium (NDSS)*, volume 16, pages 1–16, 2016.

[52] Minghua Wang, Purui Su, Qi Li, Lingyun Ying, Yi Yang, and Dengguo Feng. Automatic polymorphic exploit generation for software vulnerabilities. In *Procedings of the 9th International Conference on Security and Privacy in Communication Systems (SecureComm)*, pages 216–233, September 2013.

[53] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium (USENIX SS)*, August 2018.

[54] Luhang Xu, Weixi Jia, Wei Dong, and Yongjun Li. Automatic exploit generation for buffer overflow vulnerabilities. In *Proceedings of the 4th IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, July 2018.

[55] Fadi Yilmaz, Meera Sridhar, and Wontae Choi. Guide me to exploit: Assisted ROP exploit generation for ActionScript virtual machine. In *Annual Computer Security Applications Conference*, pages 386–400, 2020.

[56] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, OCt-Nov 2017.

[57] Awad Younis, Yashwant K Malaiya, and Indrajit Ray. Assessing vulnerability exploitability risk using software properties. *Software Quality Journal*, 24(1):159–202, 2016.

[58] Ming Yuan, Ye Li, and Zhoujun Li. Hijacking your routers via control-hijacking urls in embedded devices with web interfaces. *Information and Communications Security (ICICS)*, 10631:363–373, 2017.