Guide Me to Exploit: Assisted ROP Exploit Generation for ActionScript Virtual Machine

Conference Paper · December 2020

DOI: 10.1145/3427228.3427568

3 authors:

Fadi Yilmaz
Ankara Yildirim Beyazit University
10 PUBLICATIONS **12** CITATIONS
SEE PROFILE

Meera Sridhar
University of North Carolina at Charlotte
28 PUBLICATIONS **136** CITATIONS
SEE PROFILE

Wontae Choi
Google Inc.
17 PUBLICATIONS **541** CITATIONS
SEE PROFILE

Guide Me to Exploit: Assisted ROP Exploit Generation for ActionScript Virtual Machine

Fadi Yilmaz
fyilmaz@uncc.edu
University of North Carolina at Charlotte
Dept. of Software and Info Systems
Charlotte, North Carolina

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

Wontae Choi *
wtchoi.kr@gmail.com

ABSTRACT

Automatic exploit generation (AEG) is the challenge of determining the *exploitability* of a given vulnerability by exploring all possible execution paths that can result from triggering the vulnerability. Since typical AEG implementations might need to explore an unbounded number of execution paths, they usually utilize a fuzz tester and a symbolic execution tool to facilitate this task. However, in the case of language virtual machines, such as the ActionScript Virtual Machine (AVM), AEG implementations cannot leverage fuzz testers or symbolic execution tools for generating the exploit script, because of two reasons: (1) fuzz testers cannot efficiently generate grammatically correct executables for the AVM due to the improbability of randomly generating highly-structured executables that follow the complex grammar rules and (2) symbolic execution tools encounter the well-known program-state-explosion problem due to the enormous number of control paths in early processing stages of a language virtual machine (e.g., lexing and parsing).

This paper presents *GuidExp*, a *guided* (semi-automatic) exploit generation tool for AVM vulnerabilities. GuidExp synthesizes an exploit script that exploits a given ActionScript vulnerability. Unlike other AEG implementations, GuidExp leverages *exploit deconstruction*, a technique of splitting the exploit script into many smaller code snippets. GuidExp receives *hints* from security experts and uses them to determine places where the exploit script can be split. Thus, GuidExp can concentrate on synthesizing these smaller code snippets in sequence to obtain the exploit script instead of synthesizing the entire exploit script at once. GuidExp does not rely on fuzz testers or symbolic execution tools. Instead, GuidExp performs exhaustive search adopting four optimization techniques to facilitate the AEG process: (1) *exploit deconstruction*, (2) *operand stack verification*, (3) *instruction tiling*, and (4) *feedback from the AVM*. A running example highlights how GuidExp synthesizes the exploit script for a real-world AVM use-after-free vulnerability. In addition, GuidExp's successful generation of exploits for ten other AVM vulnerabilities is reported.

CCS CONCEPTS

• **Security and privacy → Web application security**; **Software security engineering**.

KEYWORDS

automatic exploit generation, ActionScript language, language virtual machines, vulnerabilities, fuzz testing, program synthesizing

ACM Reference Format:
Fadi Yilmaz, Meera Sridhar, and Wontae Choi . 2020. Guide Me to Exploit: Assisted ROP Exploit Generation for ActionScript Virtual Machine. In *Annual Computer Security Applications Conference (ACSAC 2020), December 7–11, 2020, Austin, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3427228.3427568

1 INTRODUCTION

Determining *exploitability* [103] of a given vulnerability, or exploit generation for that vulnerability, has historically been a labor-intensive manual process requiring deep security knowledge. However, with the recent advances in fuzz testing and symbolic execution, several approaches for automatically generating exploits have been proposed [1, 8, 11, 14, 20, 30, 31, 34, 38, 39, 41, 42, 49, 59, 60, 73, 82, 89, 92, 94, 99, 100, 102, 104]. These approaches, collectively known as the field of *automatic exploit generation* (AEG), (such as AEG for *return-oriented programming*, or *control-flow hijacking*) are critical for auditing software security, and attack prevention.

AEG implementations are usually driven by one of two engines: a *fuzzer* [64] and a *symbolic execution tool* [50]. 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 [44, 80]. 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. 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.

*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.

ACSAC 2020, December 7–11, 2020, Austin, USA

ACM ISBN 978-1-4503-8858-0/20/12…$15.00
https://doi.org/10.1145/3427228.3427568

In this work, we focus on exploit generation targeting vulnerabilities in language VMs, specifically the ActionScript Virtual Machine (AVM) [6]. The AVM is a major component of the internet ecosystem. Over the last five years more than 700 vulnerabilities were discovered in the AVM versions. In 2016, ActionScript (AS) vulnerabilities were the primary vehicle for web-based ransomware and banking trojans, accounting for ~80% of successful Nuclear exploits [29] and six of the top ten exploit kit vulnerabilities [81]. More recently, in 2018 and 2019, four zero-day exploits, CVE-2018-4878,15982 [68, 70], and CVE-2019-8069,8070 [71, 72] were discovered. In addition, National Vulnerability Database (NVD) rated the severity of 14 AVM vulnerabilities [65], discovered in the last two years, at 9.8 out 10 and identified them as critical [69].

We present 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 that reaches the exploit program state into many shorter paths. 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*.

An exploit subgoal declares a specification for synthesizing a code snippet that performs a malicious activity such as `'having a corrupted memory space'`. Execution of the code synthesized for one exploit subgoal sets the stage for executing the next exploit subgoal. After synthesizing all code snippets, GuidExp stitches the code snippets that achieve exploit subgoals together to obtain the exploit script.

Unlike the other AEG implementations, GuidExp adopts several different principles. First, GuidExp aims to reach only one exploited program state decided by the security expert. Second, GuidExp focuses on producing the exploit script whose execution reaches the exploited program state with the shortest execution path since it explores the execution-path-space as level-order. Third, GuidExp ensures that the execution of the exploit script goes through all program states given by security expert that are used to split the exploit script into smaller code snippets.

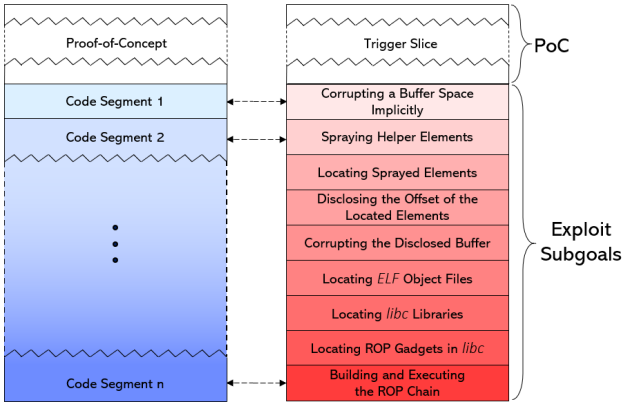
Unlike typical fuzzers, which explore execution paths by randomly mutating the given seed input (in our case the seed input is the *proof-of-concept* (PoC) exploit, which is the minimal executable that triggers the vulnerability), GuidExp generates exploit scripts by not only mutating instruction sequences inside the given PoC, but also modifying the PoC's metadata, which identifies names and parameters. Note that modifying the instruction sequence in the PoC requires modifying the metadata to allow the AVM to correctly interpret the new, modified instruction sequence. Otherwise, the AVM will not be able to parse the mutated exploit scripts and would drop them since they would not be grammatically correct. Modifying instructions inside the PoC may require making several changes to the metadata. Metadata modification includes, but not limited to, increasing the length of the function in which instructions are inserted, changing return type of a function. Therefore, GuidExp guarantees the coherence between the metadata and the instruction sequence of exploit scripts it generates.

In this paper, we focus on generating *Return-Oriented Programming* (ROP) [86] attack scripts, and demonstrate such an attack for an AVM vulnerability that we use as our running example. 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.

The contributions and impacts of our work are as follows:

- To our knowledge, we build the first guided (semi-automatic) exploit generation tool, GuidExp, targeting vulnerabilities residing in the implementation of language virtual machines, specifically AVM, which run highly-structured binaries.
- We present *exploit deconstruction*, a strategy of splitting exploit scripts that GuidExp produce into smaller code blocks. Thus, GuidExp aims to generate these smaller code blocks in sequence rather than the entire exploit at once. In our running example, we show that exploit deconstruction can reduce the complexity of AEG process by a factor of 10^{45}.
- We outline a detailed running example where we synthesize the exploit script, which performs an ROP attack, for a real-world AVM use-after-free vulnerability. In addition, we report on the production of exploit scripts for ten other real-world AVM vulnerabilities.
- Alongside exploit deconstruction, we utilize three other optimization techniques, (1) *operand stack verification*, (2) *instruction tiling*, and (3) *feedback from the AVM*, to facilitate the exploit generation process. We report that in our running example, these techniques reduce the complexity of the process by a factor of 81.9, $10^{13.5}$ and 2.38 respectively.

The rest of the paper is organized as follows. Section 2 describes overview and our technical approach. Section 3 presents implementation details of GuidExp including our running example. Section 4



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

Figure 1: Exploit Structures

introduces our optimization techniques, and Section 5 outlines experimental results. Section 6 discusses the security analysis of our approach, and design challenges. Sections 7 and 8 outline related work and the conclusion, respectively.

2 OVERVIEW

2.1 Structure of a Typical ROP Attack

In this section, we introduce the structure of a typical ROP attack. GuidExp uses ROP attacks as representative attacks, because since 2015 almost 80% (547/698) of disclosed ActionScript (AS)1 vulnerabilities could lead to an arbitrary code execution by implementing an ROP attack [66]. Therefore, we ensure that GuidExp is expected and capable of generating exploit scripts that perform an ROP attacks.

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* [32] (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 will generate 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. Our tool aims to generate real exploit code that achieves the above by appending generated code to the bug-triggering code (the trigger 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 it performs no further activity so as not to raise kernel panic. For a given vulnerability, GuidExp will use the same trigger slice as a prefix to an entire set of executables to be tested for potential exploit candidacy, therefore it is important that the trigger slice avoids kernel panic, since otherwise, the generated executables will 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 will be used by GuidExp to synthesize code blocks that will achieve that particular semantic goal. Together, the series of subgoals will produce code that will 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 [85].

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 § 3.1 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 will take 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 [6]. 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 will determine 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; var secondVariable = 12;}
```

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`) will be 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 will test 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. GuidExp 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 synthesizes. Subsequently, GuidExp moves to the next subgoal. The checkpoint that achieves the given exploit subgoal for the example in §2.3 is:

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

Acquiring a checkpoint successfully enables the exploit to be ready to aim for the next exploit subgoal; therefore, GuidExp can stitch the exploit script from checkpoints it synthesizes. When achieving one subgoal and synthesizing the next one, GUIDEEXP builds candidates for the next subgoal on top of the solution found for the previous one (i.e., new instruction permutations are appended to a solution for the previous subgoal). This guarantees that the solution for the new subgoal always satisfies the prior subgoal.

2.5 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

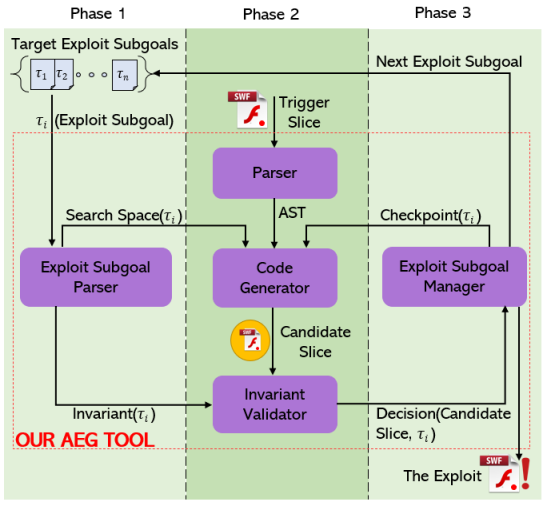


Figure 2: Overview AEG Tool

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 terminate on a red node, since no matter what opcode is appended to the candidate slice, its execution raises the same error (see § 4.4).
(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 [5]. 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.

2.6 Building the ROP Chain

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.

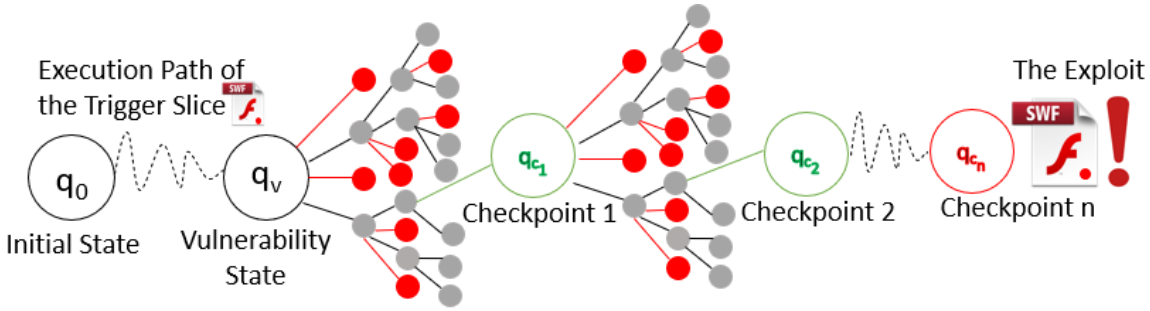


Figure 3: Exploit Script Generation Process

The ROP chain GuidExp builds has the same gadget order with a ROP chain generated by the tool called ROPgadget [46], which also builds an ROP chain from gadgets that it locates and that can be accessed during the execution of given binary.

3 IMPLEMENTATION

In this section, we introduce an example of vulnerability and walk the reader through how GuidExp produces the exploit script for this vulnerability.

3.1 Target Vulnerability

CVE-2015-5119 vulnerability 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 [48].

```
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| public class hClass {
8|   private var b2 = 0;
9|   public function hClass(var b3) {b2 = b3;}
10|  public function valueOf() {
11|    b2.length = 0x400;
12|    return 0x40;}
13| }
```

Listing 1: The PoC 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 [7]. A zero-day attack abusing this vulnerability was discovered after the attackers stole 400GB of confidential company data and made them publicly available [51]. 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 10, 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.

3.2 Preparation: Defining Exploit Subgoals, Inputs & Outputs

As mentioned in §2.3, GuidExp takes as input a collection of exploit subgoals and outputs the exploit script if the target vulnerability is exploitable. In this section, we discuss the details of the inputs that the security experts need to provide to GuidExp in order to get the exploit script that performs an ROP attack. While in practice GuidExp takes all exploit subgoals as input at the beginning of the exploit generation process, for simplicity, here we discuss this process only in the context of the first exploit subgoal.

In our running example, the first target exploit subgoal in a typical ROP attack, as shown in Fig. 1b, is '`Corrupting a Buffer Space Implicitly`'. ROP attacks need to obtain a buffer space to locate reusable code segments in sensitive system resources such as `libc` or `ELF` binaries through this exploit subgoal. The exploit subgoal can be achieved by appending the trigger slice with the following source code:

```
1| Exploit.collection.push(new Vector<uint>(0x200))
```

This line of code creates a `Vector` instance with length `0x200` that accepts only `uint` (unsigned integers) elements. The `Vector` instance is assigned to the memory chunk previously freed with the malicious function calls in Listing 1 since the garbage collector works with "last-in, first-out" principle and the memory chunk is the last element freed by the AVM.

In order to synthesize this source code, we first need to provide a search space. A search space consists of the opcodes and parameters in the equivalent bytecode representation of this source code. The optimal bytecode representation here consists of twelve opcode-parameter pairs, and includes nine different AS bytecode opcodes [6, 84]: `getglobalscope`, `getslot`, `getproperty`, `setproperty`, `findpropstrict`, `pushshort`, `applytype`, `construct`, `callproperty` and six different parameters; the constant pool indices of the `Strings` `Exploit`, `collection`, `uint`, `Vector` , `push`, and the value of `0x200`.

The invariant for the first exploit subgoal should test whether a candidate slice corrupts the `length` property of a `Vector` instance.

To do that, the invariant should check the length of currently allocated `Vector` instances, and return `true` if one of the `Vector` instances have a length of a large number, such as `0x10000000`. An optimal invariant to perform this check is the following:

```
1| for(var i=0; i< Exploit.collection.length; i++){
2|   if(Exploit.collection[i].length > 0x10000000){
3|     _corrupted = Exploit.collection[i];
4|     return true;}}
```

Line 1 iterates over all `Vector` instances allocated during the execution of each candidate slice and Line 2 checks whether any of the `Vector` instances has a length greater than `0x10000000`. Line 3 holds the corrupted `Vector` instance to be used in later stages of the exploit. Line 4 returns `true` and is reached only if such a corrupted `Vector` instance is created.

In this example above, we expect the security expert to specify a sequence of the exploit subgoals, in which the first exploit subgoal consists of the search space and the invariant we mentioned above, and the PoC to allow GuidExp to know the execution path in which the vulnerability is triggered.

3.3 Phase 1: Exploit Subgoal Processing

In the first phase, GuidExp reads the first exploit subgoal and produces the corresponding search space and invariant. The search space is sent to the Code Generator and the invariant is the input for the Invariant Validator. For the first exploit subgoal, GuidExp reads the search space and the invariant mentioned in §3.2.

3.4 Phase 2: Generating Candidate Slices and Validating Invariant

In our running example, the code generator must make use of the dangling pointer, which occurs after the trigger slice is executed. The dangling pointer points to the `length` property, which is a 32-bit value, of the subsequently created `Vector` instance, which enables the exploit to corrupt the `length` property by using `b1`. As the modern computer architecture adopts `little-endian` format, the index 3 of `b1` corresponds to the most significant byte. Thus, the exploit code corrupts the `length` property of the `Vector` instance by replacing Line 6 of Listing 1 with the following source code:

```
6| b1[3] = mal;
```

This overrides the most significant byte of the `length` property and its new value becomes `0x40000200`. Therefore, the exploit can access any memory chunk in the memory that running AVM instance can access during its execution.

The Invariant Validator receives the invariant from `Phase 1` and is responsible for inserting the invariant for the current exploit subgoal into candidate slices that the Code Generator generates.

In our running example, the Invariant Validator injects the invariant that it receives from the first exploit subgoal, into the candidate slice. Since the Invariant Validator does not know which `Vector` instance will be corrupted, it must be supported with a `global` and `static` `Vector` collection. GuidExp automatically checks all instance creation in candidate slices and if the created instance type is `Vector`, GuidExp pushes the `Vector` instance to the `global` `Vector` collection. The Invariant Validator therefore can keep track of all `Vector` instances created during the execution of the candidate slice. In addition, Line 3 of the optimal invariant stores the corrupted `Vector` instance if a candidate slice succeeds to create one. The Invariant Validator modifies the trigger slice to add the definition of the collection to `global` scope.

3.5 Phase 3: Evaluating Candidate Slices

In our running example, the candidate slice, "Checkpoint(τ_1)", that satisfies the first exploit subgoal is given in Listing 2. Lines highlighted in green in Listing 2 are inserted by the Invariant Validator, and lines highlighted with light blue in Listing 2 are generated and inserted by the Code Generator. GuidExp uses "Checkpoint(τ_1)" as the skeleton code for synthesizing "Checkpoint(τ_2)". Note that when Checkpoint(τ_1) is used as a skeleton, the code injected by Invariant Validator will be ignored.

GuidExp performs the same procedure with "Checkpoint(τ_1)" and the second exploit subgoal to find Checkpoint(τ_2). GuidExp repeats this until all exploit subgoals are achieved. When all target exploit subgoals are achieved, the Exploit Subgoal Manager outputs the exploit code. Thus, security engineers can analyze the exploit code to see how the target vulnerability is exploited, and how the exploit code uses the vulnerability to perform an actual attack against their security protections.

```
1| public class malClass extends Sprite {
2|   public function malClass() {
3|  protected static var collection:* = new Vector.
4|  <Vector.<uint»();
5|    var b1 = new ByteArray();
6|    b1.length = 0x200;
7|    var mal = new hClass(b1);
8|  b1[3] = mal;
9|  Exploit.collection.push(new Vector<uint>(0x200));
10|  for(var i=0; i< Exploit.collection.length; i++){
11|  if(Exploit.collection[i].length > 0x10000000)
12|  return true;}}
```

Listing 2: Source code representation of `malclass` in Checkpoint(τ_1)

4 OPTIMIZATION TECHNIQUES

Finding the correct permutation of instructions given in exploit subgoals requires testing all possible permutations in the worst case. As mentioned in §3.2, in our running example, the 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 test all possible permutations, which is not practical. In this section, we discuss four optimization techniques that we successfully implemented to address this challenge and reduce the number of candidate slices to be tested, leveraging language features of the AS language.

Table 1 demonstrates the efficiency results for our optimization techniques for our running example. Rows are labeled with numbers given in parentheses. The left cell of a row describes the value given in the corresponding right cell. Rows written in bold show the effectiveness of our optimization techniques and having exploit subgoals. If a value requires to be calculated, the calculation formula is given in the same cell, below the value. Numbers in parentheses used in these calculations refer to the value of the corresponding rows.

4.1 Deconstructing an Exploit into Subgoals

As mentioned in §2.3 and demonstrated in Fig. 1b, GuidExp splits the target exploit script into many smaller exploit subgoals in order to facilitate the exploit generation task. This is our first optimization technique, and we refer to this henceforth as *exploit deconstruction*.

With exploit deconstruction, GuidExp targets synthesizing exploit subgoals in sequence instead of synthesizing the entire exploit script at once. Therefore, GuidExp can define a checkpoint for every exploit subgoal on the execution path of the 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. Fig. 3 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. For our running example, the efficiency of exploit deconstruction technique for synthesizing Checkpoint(τ_1) and Checkpoint(τ_2) is given in the eighth row of Table 1.

4.2 Operand Stack Verification

Computation in the AVM is based on executing the code sequence of method bodies, the constant pool, and the heap for non-primitive data objects created at run-time. The code sequence is composed of instructions. Each instruction modifies the state of the AVM or has an effect on the run-time environment by means of input or output. To manage the execution of method bodies, the AVM employs an *operand stack*, which holds operands for the instructions and stores their results. The scope stack is part of the run-time environment and stores objects that are to be searched by the AVM.

Since GuidExp generates a candidate slice for every permutation of instructions given in the search spaces, some candidate slices could perform illegal operand stack operations. These illegal operations can cause two types of errors related to the operand stack: (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 our second optimization technique, *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 because they will raise the same error regardless of instructions they add to an `ill-prefix` permutation. 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. As shown in the sixteenth row of Table 1, GuidExp can disqualify 98.78% of the generated candidate slices by using the operand stack verification technique for our running example.

Description	Value
(1) Number of opcodes in AS language	164 [84]
(2) Number of parameters in our trigger slice	33
(3) Number of instructions needed to append to the trigger slice to produce Checkpoint(τ_1)	12
(4) Number of instructions needed to append to the Checkpoint(τ_1) to produce Checkpoint(τ_2)	23
(5) Number of candidate slices that GuidExp needs to generate to produce Checkpoint(τ_1)	$164^{12} * 33^{12}$ $(1)^{(3)} * (2)^{(3)}$
(6) Number of candidate slices that GuidExp needs to generate to produce Checkpoint(τ_2)	$164^{35} * 33^{35}$ $(1)^{(3)+(4)} * (2)^{(3)+(4)}$
(7) Number of candidate slices that GuidExp needs to generate to produce Checkpoint(τ_2) with exploit deconstruction	$164^{12} * 33^{12} + 164^{23} * 33^{23}$ $(5) + (1)^{(4)} * (2)^{(4)}$
(8) **Efficiency of exploit deconstruction for the first exploit subgoal**	$\approx 10^{45}$ $(6)/(7)$
(9) Number of candidate slices that GuidExp needs to generate to produce Checkpoint(τ_1) by utilizing the first exploit subgoal	54^{12} (please see §4)
(10) **Efficiency of restricting the search space in the first exploit subgoal**	$\approx 10^{24}$ $(5)/(9)$
(11) Number of tiles in the first subgoal	8
(12) **Efficiency of instruction tiling for the exploit subgoal**	$\approx 10^{13.5}$ $(9)/(11)^{(11)}$
(13) Number of candidate slices GuidExp needs to generate to satisfy the first exploit subgoal	2,396,744 $\sum_{n=1}^{(11)-1}(11)^n$
(14) Number of candidate slices that pass the operand stack verification	29,167
(15) Number of candidate slices that pass the operand stack verification and feedback from the AVM	12,229
(16) **Percentage of candidate slices discarded by the operand stack verification for the first exploit subgoal**	98.78% $1 - (14)/(13)$
(17) **Percentage of candidate slices discarded based on the feedback from the AVM for the first exploit subgoal**	58% $1 - (15)/(14)$

Table 1: Efficiency calculation of optimization techniques

4.3 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 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.

4.4 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 stores `ill-prefix` permutations in a search tree so that it can quickly decide whether future candidate slices contain an `ill-prefix`. Therefore, the Code Generator discards subsequently generated candidate slices if they contain an `ill-prefix` permutation, since instruction sequences are prefix-closed, and will raise the same error. As shown in the seventeenth row of the Table 1, in experiments with our running example, by using the feedback from the AVM, GuidExp discards 58.07% of the candidate slices.

5 EXPERIMENTAL RESULTS

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` [67]. GuidExp was written in Java with NetBeans IDE 8.0.2 JDK v.1.8.

We synthesized exploit scripts for eleven different AVM vulnerabilities, including our running example vulnerability, CVE-2015-5119. We selected these vulnerabilities because these vulnerabilities were frequently used in famous exploit kits such as Nuclear [29], Neutrino [48], Angler [48], Gong Da [83], and Cool [83]. In addition, these vulnerabilities are well-publicized so that we can create the corresponding exploit subgoals for these vulnerabilities accurately.

We conducted two set of experiments. In the first set, GuidExp utilized an open-source core implementation of the AVM, `avmplus` [3] provided by Adobe, to execute candidate slices GuidExp generated for our running example vulnerability. Table 2 demonstrates our experimental results with the open-source core implementation of the AVM for our running example vulnerability. We give the number of generated candidate slices and executed candidate slices during synthesizing each exploit subgoal. GuidExp outputs the exploit script within slightly below 15 minutes.

To our knowledge, the open-source core version contains only one vulnerability which is our running example vulnerability. Assuming that the security experts would have the source of their application, we highlight the performance of GuidExp with the open-source version. However, to demonstrate the generality of GuidExp, we conducted the second set of experiments with the closed-source Flash Player v11.2.202.262 [2] because this particular Flash Player contains all eleven vulnerabilities we selected. Table 2 also shows our experimental results with the closed-source Flash Player, v11.2.202.262, for our running example vulnerability. Running a closed-source AVM brings two interesting changes. First, exploit generation process takes around 45 times longer compared to our first set of experiments.

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 $\sim$ 89% of the time required to test one candidate slice, producing the slice takes $\sim$ 1%, executing the slice takes $\sim$ 6%, reading the result takes $\sim$ 4%. However, the open-source version's initialization overhead is smaller. The initialization overhead only takes $\sim$ 11% of the experiment for open-source AVM. With the applications that their initialization and termination take significantly less time, the overhead of using the closed-source version of these applications should not affect the performance of our tool dramatically. 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. However, 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.

Table 3 shows our experimental results with eleven other 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. Additionally, we demonstrate that GuidExp can tolerate some level of inaccuracy (providing larger search spaces) in defining exploit subgoals to allow security experts to have some space. Please see the Appendix for details.

6 DISCUSSION

Challenges. One of the biggest challenges we faced in this work is that the PoCs that we found online were not compatible with the open-source AVM implementation [3]. In addition, some PoCs that we found were written for different versions of the AVM, which adopt different memory layouts for the run-time instances. Therefore, we crafted our PoCs by tailoring these PoCs. We recalculated offsets of the attributes used for triggering vulnerabilities, removed external libraries used in the PoCs or if their source is publicly available, we statically compiled these libraries with the core implementation to include them.

Since we provide the exploit subgoals to GuidExp to conduct our experiments, we need to obtain a deep understanding of how a ROP

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

attack exploits given vulnerabilities and bypasses modern operating system security mechanisms such as ASLR or DEP. However, the PoCs we craft and the exploit code GuidExp synthesizes perform their malicious activities implicitly. For example, the exploit script corrupts the length of the `Vector` instance without calling the `.length` property, but with exploiting the unusual situation of the AVM that occurs after triggering the vulnerability. Therefore, we could not use any AS debugger to observe run-time behaviors of the exploit code to understand how the exploit script tricks the AVM to perform its malicious intention surreptitiously. Hence, we utilized `GNU debugger` [35] to debug the AVM and observe run-time behaviors of the exploit code by scrutinizing the memory cells to see how the instructions modify memory cells.

Limitations. GuidExp's performance strictly depends on the accuracy of the exploit subgoals. Having redundant instructions in an exploit subgoal significantly increases the time that GuidExp needs to generate the exploit script, since the number of permutations of instructions increases exponentially as the number of instructions increases linearly. The number of executed candidate slices works as a coefficient for the performance of our tool since the tool applies the same approach to all of the executed candidate slices. We mentioned the factors that affect the performance of our tool in §5.

Use Cases & Effort. Our exploit generation approach does not leverage a typical fuzz tester or symbolic execution tool due to the reasons discussed early in §1. Instead, we leverage the coherence between *tool-centered, human-assisted* [88] vulnerability analysis paradigm, which leverages human expertise to break the execution-path-space into relatively smaller search spaces and the pattern that ROP attacks follow.

Table 3: Exploit generation 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

The primary target audience of GuidExp is security experts who have thorough knowledge about the AVM and the AS language. Examples of such users include industry security testers, academics, defense contractors, and government laboratories. GuidExp uses the subgoals to narrow down the execution-path space, therefore, the performance and success of GuidExp depend on the quality of provided subgoals. However, a newbie can also benefit by using GuidExp to synthesize simple exploits (GuidExp is not limited to ROP) or achieve subgoals iteratively. The AEG process will take longer with less accurate subgoals (i.e., larger search space), as the size of the search space increases combinatorial rate.

GuidExp needs to be given accurate exploit subgoals to produce the exploit script. This raises two important questions:

Q1: To what extent can GuidExp help craft exploits?

Even a senior security engineer can benefit significantly from GuidExp, since deciding the exploitability of a vulnerability is not a trivial task. We do believe that most of the time, the expert might not create the exploit script immediately after disclosing a vulnerability that extends the vulnerability triage process since the expert cannot decide whether the vulnerability is exploitable. For example, the subgoal "`Disclosing the Address of the Corrupted Vector`" can be achieved by appending following code to the related Checkpoint: `uv[0]=uv[pos-5]-((pos-5)* 4)-0xc`. To write this code, the expert must calculate all the offsets and indices accurately. However, when using GuidExp, it is sufficient to provide a simple search space ("uv, pos, [], subtract, multiply"), to produce the code given above. Additionally, GuidExp can detect unexpected exploit pathways, which the expert may not be familiar with, an essential prerequisite for staying ahead of attackers.

Q2 : How much manual effort is required to create search spaces?

To measure the manual effort required to use GuidExp, we first selected a UAF Flash vulnerability of which we found the open-source exploit script for. We analyzed the exploit script to understand its methods, and to deconstruct it. Then, we chose another UAF Flash vulnerability and used GuidExp to synthesize an exploit script for this vulnerability. We aim to synthesize the exploit script that exploits the second UAF Flash vulnerability with the methods we learned from the exploit script we analyzed. We consider our

expertise level "*medium*" since we are somewhat familiar with the exploit pattern and the AS bytecode language. It took about an hour to provide the subgoals (ignoring the time GuidExp takes) to produce the exploit script for the second vulnerability by using GuidExp. Without having a comprehensive user study it is hard to judge the usefulness of our tool. Therefore, as a future work, we plan to set up a detailed user study to accurately measure the level of required expertise and the correlation between the level of the expertise and required manual effort to synthesize exploit scripts by using GuidExp. We plan to publish the results in future work.

Future Directions. Our AEG approach allows future improvements and additional automation for GuidExp . For example, GuidExp can be used in conjunction with an AVM-specific fuzzer to do vulnerability detection, subsequently the exploitation for disclosed AVM vulnerabilities. In addition, currently, the code generation phase is done in a brute force manner, that is, instructions are added to existing slices until a solution is found. The search space is reduced by allowing the security experts to pass in opcodes. This makes further automation possible here: as the semantics of opcodes is known, *Constraint Logic Programming* [43] (CLP) can be used to automatically select the opcodes. GuidExp would still receive checkpoints and invariants from the security experts, but defining instructions for search spaces would require less human interaction.

7 RELATED WORK

The AEG problem is first proposed in [10], and several works [1, 8, 11, 14, 20, 30, 31, 34, 38, 39, 41, 42, 49, 59, 60, 73, 82, 89, 92, 94, 99, 100, 102, 104] address it for various types of vulnerabilities. AEG tools combine high performance fuzzing and symbolic execution to first identify software vulnerabilities and then to exploit them in an autonomous fashion. Symbolic execution tools such as SAGE [37], KLEE [19], BitFuzz [18], S2E [28], and FuzzBall [62] concentrate on searching execution paths but not generating exploits. Hybrid concolic testers [14, 52, 99] advance AEG tools by interleaving random testing with concolic execution [61]. These tools reduce and prioritize execution paths that their symbolic execution tool needs to explore. GuidExp differs from typical AEG implementations by not leveraging a fuzzer or symbolic execution tool, since these approaches are not immediately helpful for synthesizing exploit scripts for AVM vulnerabilities.

Compiler testing techniques that do automated test input generation [17, 21–23, 25, 26, 40, 53, 54, 58, 63, 87, 101, 105] are similar to GuidExp in that they also test programs that take as input other programs. These compiler testing techniques generate either diversified set of inputs based on highly-specified generation rules or new inputs by mutating existing inputs. The main difference between compiler testing techniques and GuidExp is the application; compiler testing techniques aim to find as many bugs as possible. They do not need to penetrate deep into the search space as long as bugs can be triggered with simple inputs. GuidExp is designed to serve users each of whom is interested in a single bug requiring a long sequence of code to exploit with an actual attack. As a result, GuidExp needs to search a relatively larger search space compared to conventional compiler testing techniques, which compels GuidExp to employ iterative searching and more intensive human-computer interaction.

Improving fuzzers has been an active field for decades. First, *black-box fuzzing* [64], a fuzzing approach in which fuzzers are not informed of the target program and treat it as a black-box, was proposed. Then, researchers put more focus on *white-box fuzzinzg*, more recent fuzzing strategy, in which the fuzzers symbolically execute the target application to gather constraints on inputs from conditional branches encountered along the execution [37]. *Black-box fuzzing* [64] and *white-box fuzzing* [37] refer to fuzzing with and without informed knowledge of the target program, respectively.

Coverage-based gray-box fuzzing (CGF) and *smart gray-box fuzzing* (SGF) are the most efficient and recent approaches for automated vulnerability discovery. A CGF randomly mutates some bits in given seed files to generate new files. In contrast to white-box approaches [33, 36, 37, 77], which suffer from high overhead due to constraint solving and program analysis, and black-box approaches [15, 24, 40, 95], which are limited due to lack of knowledge about target applications, CGFs utilize lightweight code mutation [12, 80, 90, 96, 97, 106]. These techniques are similar to GuidExp as it also employs lightweight code mutation based on inputs it receives. libFuzzer [97], AFL [96] and its extensions [9, 12, 13, 27, 55, 56, 74, 75, 92] constitute the most widely-used implementations of CGF. SGF leverages a high-level structural representation of the seed file to generate new files and is introduced as AFLSmart [78]. Although gray-box fuzzing embodiment can generate distinct executables to guide the fuzzer to new code regions, they are not capable of efficiently generating grammatically valid AVM scripts due to the high complexity of grammar rules adopted by the AVM.

ActionScript security has been studied extensively in the past, including inconsistencies of security models of AS and JavaScript for cross-platform web contents [79], interaction between AS and DOM *capability tokens* [57], mitigation techniques for AS-based DNS rebinding [45] and ROP attacks [76], malicious Flash URL redirections [93], anomaly-based Flash malware detection [47, 98], and lack of secure and airtight implementation of the AVM [91]. To our knowledge unlike our paper, no related work focuses on generating exploits for AS.

8 CONCLUSION

We have presented the first guided (semi-automatic) AEG tool, GuidExp, for AVM. We also showed that GuidExp can succesfully produces ROP exploit scripts given vulnerabilities in the AVM by exploring all execution paths that triggering these vulnerabilities can lead to. Unlike the other AEG tools, GuidExp does not employ a fuzz tester or a symbolic execution tool because they are not efficiently helpful since (1) fuzz testers cannot efficiently generate grammatically correct executables for the AVM due to the improbability of randomly generating highly-structured executables that follow the complex grammar rules that the AVM enforces, and (2) symbolic execution tools encounter the program-state-explosion problem due to the enormous number of control paths in early processing stages of binaries executed by the AVM.

GuidExp adopts several optimization techniques to facilitate the AEG process: (1) exploit deconstruction, which breaks the exploit

script that GuidExp synthesizes into several smaller subgoals, (2) operand stack verification, (3) instruction tiling, and (4) feedback from the AVM. GuidExp receives *hints* from security experts and it uses them to determine places where the exploit script can be split so that GuidExp can concentrate on synthesizing these subgoals in sequence instead of the entire exploit code at once. We report that these techniques reduce the complexity of the process by a factor of 10^{45}, 81.9, $10^{13.5}$, and 2.38 respectively, for our running example. In addition to our running example, we report on GuidExp-produced exploit scripts for ten other well-publicized AVM vulnerabilities.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Anno Accademico. 2013. *Static Detection and Automatic Exploitation of Intent Message Vulnerabilities in Android Applications.* Master's thesis. Politecnico Di Milano.
[2] Adobe, Inc. [n.d.]. Archived Flash Player versions. https://helpx.adobe.com/flash-player/kb/archived-flash-player-versions.html.
[3] Adobe, Inc. [n.d.]. avmplus. https://github.com/adobe/avmplus.
[4] Adobe, Inc. [n.d.]. Download Adobe Flex SDK. https://www.adobe.com/devnet/flex/flex-sdk-download.html.
[5] Adobe, Inc. [n.d.]. Run-Time Errors. https://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/runtimeErrors.html.
[6] Adobe, Inc. 2007. ActionScript Virtual Machine 2 (AVM2) Overview. https://www.adobe.com/content/dam/acom/en/devnet/pdf/avm2overview.pdf.
[7] Adobe, Inc. 2015. Adobe Security Bulletin. http://tinyurl.com/ofdwo9c. Accessed" 2016-12-03.
[8] Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and V.N. Venkatakrishnan. 2016. Chainsaw: Chained Automated Workflow-based Exploit Generation. In *Proceedings of the 23th ACM Conference on Computer and Communications Security (CCS)*.
[9] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Proceedings of The Network and Distributed System Security Symposium (NDSS)*, Vol. 19. 1–15.
[10] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. 2011. AEG: Automatic Exploit Generation. In *Proceedings of The Network and Distributed System Security Symposium (NDSS)*.
[11] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking Blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP'14)*.
[12] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2329–2344.
[13] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering* 45, 5 (2017), 489–506.
[14] Konstantin Böttinger and Claudia Eckert. 2016. DeepFuzz: Triggering Vulnerabilities Deeply Hidden in Binaries. *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)* 9721 (2016), 25–34.
[15] Sergey Bratus, Axel Hansen, and Anna Shubina. 2008. LZfuzz: a fast compression-based fuzzer for poorly documented protocols. *Darmouth College, Hanover, NH, Tech. Rep. TR* 634 (2008).
[16] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. 2008. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*. ACM, 27–38.
[17] Colin J Burgess and M Saidi. 1996. The automatic generation of test cases for optimizing Fortran compilers. *Information and Software Technology* 38, 2 (1996), 111–119.
[18] Juan Caballero, Pongsin Poosankam, Stephen McCamant, Dawn Song, et al. 2010. Input generation via decomposition and re-stitching: Finding bugs in malware. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*. 413–425.
[19] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation ((OSDI))*, Vol. 8. 209–224.
[20] Dan Caselden, Alex Bazhanyuk, Mathias Payer, Stephen McCamant, and Dawn Song. 2013. HI-CFG: Construction by Binary Analysis and Application to Attack Polymorphism. In *Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS)*, Jason Crampton, Sushil Jajodia, and Keith Mayes (Eds.). 164–181.
[21] Junjie Chen. 2018. Learning to Accelerate Compiler Testing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18)*. 472–475.
[22] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. 2017. Learning to prioritize test programs for compiler testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 700–711.
[23] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. Test Case Prioritization for Compilers: A Text-Vector Based Approach. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*.
[24] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *Proceedings of the 26th Annual Network & Distributed System Security Symposium (NDSS)*.
[25] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. Compiler Bug Isolation via Effective Witness Test Program Generation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. 223–234.
[26] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and XIE Bing. 2018. Coverage Prediction for Accelerating Compiler Testing. *IEEE Transactions on Software Engineering* (2018).
[27] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy ((SP))*. 711–725.
[28] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 265–278.
[29] Cisco. 2016. Cisco 2016 Midyear Security Report. https://tinyurl.com/y7kupmkr.
[30] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. 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)*. 952–963.
[31] Jared D. DeMott, Richard J. Enbody, and WIlliam F. Punch. 2011. Towards an Automatic Exploit Pipeline. In *Proceedings of the 6th International Conference for Internet Technology and Secured Transactions (ICITST)*.
[32] Pavel Dovgalyuk, Denis Dmitriev, and Vladimir Makarov. 2015. Don't panic: reverse debugging of kernel drivers. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*.
[33] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering, (ICSE)*. 474–484.
[34] Joshua Garcia, Mahmoud Hammad, Negar Ghorbani, and Sam Malek. 2017. Automatic Generation of Inter-Component Communication Exploits for Android Applications. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*.
[35] GNU. 2019. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/.
[36] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-based whitebox fuzzing. *ACM Sigplan Notices* 43, 6 (2008), 206–215.
[37] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of the 16th Annual Network & Distributed System Security Symposium (NDSS)*.
[38] Sean Heelan. 2011. *Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities.* Master's thesis. University of Oxford.
[39] Sean Heelan, Tom Melham, and Daniel Kroening. 2018. Automatic Heap Layout Manipulation for Exploitation. In *Proceedings of the 27th USENIX Security Symposium (USENIX SS)*.
[40] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*. 445–458.
[41] Hu Hong, Chua Zheng Leong, Adrian Sendroiu, Saxena Prateek, and Liang Zhenkai. 2015. Automatic Generation of Data-oriented Exploits. In *Proceedings of the 24th USENIX Conference on Security Symposium (USENIX SS)*. 177–192.
[42] Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang, Chung-Wei Lai, Han-Lin Lu, and Wai-Meng Leong. 2012. 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*. IEEE, 78–87.
[43] J. Jaffar and J.-L. Lassez. 1987. Constraint Logic Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming*

Languages (Munich, West Germany) *(POPL '87)*. Association for Computing Machinery, New York, NY, USA, 111–119. https://doi.org/10.1145/41625.41635

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

[45] Martin Johns, Sebastian Lekies, and Ben Stock. 2013. Eradicating DNS Rebinding with the Extended Same-origin Policy. In *Proc. of the 22nd USENIX Security Symp. (SS)*. 621–636.

[46] JonathanSalwan. [n.d.]. ROPgadget. https://github.com/JonathanSalwan/ROPgadget.

[47] Wookhyun Jung, Sangwon Kim, and Sangyong Choi. 2015. Poster: Deep Learning for Zero-day Flash Malware Detection. http://tinyurl.com/zvqpvfl.

[48] Kaspersky [n.d.]. Kaspersky Security Bulletin 2015. The overall statistics for 2015. http://tinyurl.com/zgkkdbj.

[49] Cha Sang Kil, Avgerinos Thanassis, Rebert Alexandre, and Brumley David. 2012. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP'12)*. 380–394.

[50] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (july 1976), 385–394.

[51] Eduard Kovacs. [n.d.]. Two New Flash Player Zero-Day Bugs Found in Hacking Team Leak. tinyurl.com/y25a6ve5.

[52] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN Conference Programming Language Design and Implementation ((PLDI))*. 193–204.

[53] Stephen Kyle, Hugh Leather, Björn Franke, Dave Butcher, and Stuart Monteith. 2015. Application of Domain-aware Binary Fuzzing to Aid Android Virtual Machine Testing. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '15)*. 12 pages.

[54] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Notices* 50, 10 (2015), 386–399.

[55] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 475–485.

[56] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (FSE)*. 627–637.

[57] Zhou Li and XiaoFeng Wang. 2010. FIRM: Capability-based inline mediation of Flash behaviors. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*.

[58] Christian Lindig. 2005. Random testing of C calling conventions. In *Proceedings of the 6th International Symposium on Automated analysis-driven debugging*. ACM, 3–12.

[59] Lannan Luo, Qiang Zeng, Chen Cao, Kai Chen, Jian Liu, Limin Liu, Neng Gao, Min Yang, Xinyu Xing, and Peng Liu. 2017. 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)*.

[60] Lannan Luo, Qiang Zeng, Chen Cao, Kai Chen, Jian Liu, Limin Liu, Neng Gao Min Yang, Xinyu Xing, and Peng Liu. 2016. Context-aware System Service Call-oriented Symbolic Execution of Android Framework with Application to Exploit Generation. *CoRR* (2016).

[61] Rupak Majumdar and Koushik Sen. 2007. Hybrid concolic testing. In *29th International Conference on Software Engineering ((ICSE'07))*. IEEE, 416–426.

[62] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 2012. Path-exploration lifting: Hi-fi tests for lo-fi emulators. *ACM SIGARCH Computer Architecture News* 40, 1 (2012), 337–348.

[63] WM McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10 (1998), 100–107.

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

[65] MITRE, Inc. [n.d.]. Common Vulnerabilities and Exposures Database. https://cve.mitre.org/. Accessed: 2018-01-24.

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

[67] Mozilla.org. [n.d.]. Tamarin Project. https://www-archive.mozilla.org/projects/tamarin/.

[68] National Institute of Standards and Technology. 2018. CVE-2018-15982. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-15982.

[69] National Institute of Standards and Technology. 2018. CVE-2018-15982 Detail. https://nvd.nist.gov/vuln/detail/CVE-2018-15982.

[70] National Institute of Standards and Technology. 2018. CVE-2018-4878. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-4878.

[71] National Institute of Standards and Technology. 2019. CVE-2019-8069 . https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-8069.

[72] National Institute of Standards and Technology. 2019. CVE-2019-8070. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-8070.

[73] V. A. Padaryan, V. V. Kaushan, and A. N. Fedotov. 2015. Automated Exploit Generation for Stack Buffer Overflow Vulnerabilities. *Programming and Computer Software* 41 (2015). Issue 6.

[74] Peng, Hui and Shoshitaishvili, Yan and Payer, Mathias. 2018. T-Fuzz: fuzzing by program transformation. In *Proceedings of the 39th IEEE Symposium on Security and Privacy ((SP))*. 697–710.

[75] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2155–2168.

[76] David Pfaff, Sebastian Hack, and Christian Hammer. 2015. *Proc. of the 7th Int. Symp. on Engineering Secure Software and Systems (ESSoS)*. Chapter Learning How to Prevent Return-Oriented Programming Efficiently, 68–85.

[77] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2016. Model-based whitebox fuzzing for program binaries. In *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 543–553.

[78] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *IEEE Transactions on Software Engineering (TSE)* (2019).

[79] Phu H. Phung, Maliheh Monshizadeh, Meera Sridhar, Kevin W. Hamlen, and V.N. Venkatakrishnan. 2015. Between Worlds: Securing Mixed JavaScript/ActionScript Multi-party Web Content. *IEEE Trans. on Dependable and Secure Computing (TDSC)* 12, 4 (2015), 443–457.

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

[81] Recorded Future. 2016. New Kit and Same Player: Top 10 Vulnerabilities Used by Exploit Kits in 2016. https://www.recordedfuture.com/top-vulnerabilities-2016/.

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

[83] Eric Romang. [n.d.]. Gong Da Exploit Pack Add Flash CVE-2013-0634 Support. https://tinyurl.com/w6l4sjw/.

[84] Michael Schmalle. [n.d.]. AS3Commons - Opcodes. https://github.com/teotigraphix/as3-commons/blob/master/as3-commons-bytecode/src/main/actionscript/org/as3commons/bytecode/abc/enum/Opcode.as. Accessed on=9-11-2019.

[85] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2011. Q: Exploit Hardening Made Easy. In *Proceedings of the 20th USENIX Security Symposium*.

[86] Hovav Shacham. 2007. 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)*. 552–561.

[87] Flash Sheridan. 2007. Practical testing of a C99 compiler using output comparison. *Software: Practice and Experience* 37, 14 (2007), 1475–1488.

[88] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. 2017. Rise of the HaCRS: Augmenting autonomous cyber reasoning systems with human assistance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 347–362.

[89] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. 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)*.

[90] Sherri Sparks, Shawn Embleton, Ryan Cunningham, and Cliff Zou. 2007. Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. In *The 23rd Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE, 477–486.

[91] Meera Sridhar, Abhinav Mohanty, Fadi Yilmaz, Vasant Tendulkar, and Kevin W. Hamlen. 2018. Inscription: Thwarting ActionScript Web Attacks From Within. In *In the proceedings of the 17th International Conference On Trust and Security and Privacy In Computing and Communications (TrustCom)*. 504–515.

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

[93] Kurt Thomas, Chris Grier, Justin Ma, Vern Paxson, and Dawn Song. 2011. Design and Evaluation of a Real-time URL Spam Filtering Service. In *Proc. of the 32nd IEEE Symp. on Security & Privacy (S&P)*. 447–462.

[94] Minghua Wang, Purui Su, Qi Li, Lingyun Ying, Yi Yang, and Dengguo Feng. 2013. Automatic Polymorphic Exploit Generation for Software Vulnerabilities. In *Procedings of the 9th International Conference on Security and Privacy in Communication Systems (SecureComm)*. 216–233.

[95] Zhiqiang Wang, Yuqing Zhang, and Qixu Liu. 2013. RPFuzzer: A Framework for Discovering Router Protocols Vulnerabilities Based on Fuzzing. *KSII Transactions on Internet & Information Systems* 7, 8 (2013).
[96] Website. [n.d.]. american fuzzy lop. http://lcamtuf.coredump.cx/afl/.
[97] Website. [n.d.]. libFuzzer: A library for coverage-guided fuzz testing. http://llvm.org/docs/LibFuzzer.html.
[98] Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2016. Comprehensive Analysis and Detection of Flash-Based Malware. (2016), 101–121.
[99] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. 2018. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium (USENIX SS)*.
[100] Luhang Xu, Weixi Jia, Wei Dong, and Yongjun Li. 2018. Automatic Exploit Generation for Buffer Overflow Vulnerabilities. In *Proceedings of the 4th IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*.
[101] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*. 283–294.
[102] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. SemFuzz: Semantics-based Automatic Generation of Proof-of-Concept Exploits. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*.
[103] Awad Younis, Yashwant K Malaiya, and Indrajit Ray. 2016. Assessing vulnerability exploitability risk using software properties. *Software Quality Journal* 24, 1 (2016), 159–202.
[104] Ming Yuan, Ye Li, and Zhoujun Li. 2017. Hijacking Your Routers via Control-Hijacking URLs in Embedded Devices with Web Interfaces. *Information and Communications Security (ICICS)* 10631 (2017), 363–373.
[105] Chen Zhao, Yunzhi Xue, Qiuming Tao, Liang Guo, and Zhaohui Wang. 2009. Automated test program generation for an industrial optimizing compiler. In *ICSE Workshop on Automation of Software Test*. 36–43.
[106] Jinjing Zhao and Ling Pang. 2018. Automated Fuzz Generators for High-Coverage Tests Based on Program Branch Predications. In *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*. 514–520.

9 APPENDIX

To demonstrate that GuidExp can tolerate some level of inaccuracy (providing larger search spaces) in defining exploit subgoals to allow security experts to have some space, we run GuidExp with 1.25, 1.5, and 2 times larger search spaces than the optimal search space (the most accurate). 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. Table 4 shows exploit script synthesizing time of GuidExp with larger spaces for CVE-2015-5119 with the open-source AVM implementation.

Table 4: 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 m 03.38s)