OCTOPOCS: Automatic Verification of Propagated Vulnerable Code Using Reformed Proofs of Concept

Seongkyeong Kwon, Seunghoon Woo, Gangmo Seong, Heejo Lee*
Korea University, {bible_kwon, seunghoonwoo, geldkang, heejo}@korea.ac.kr

***Abstract*—Addressing vulnerability propagation has become a major issue in software ecosystems. Existing approaches hold the promise of detecting widespread vulnerabilities but cannot be applied to verify effectively whether propagated vulnerable code still poses threats. We present OCTOPOCS, which uses a reformed Proof-of-Concept (PoC), to verify whether a vulnerability is propagated. Using context-aware taint analysis, OCTOPOCS extracts crash primitives (the parts used in the shared code area between the original vulnerable software and propagated software) from the original PoC. OCTOPOCS then utilizes directed symbolic execution to generate guiding inputs that direct the execution of the propagated software from the entry point to the shared code area. Thereafter, OCTOPOCS creates a new PoC by combining crash primitives and guiding inputs. It finally verifies the propagated vulnerability using the created PoC. We evaluated OCTOPOCS with 15 real-world C and C++ vulnerable software pairs, with results showing that OCTOPOCS successfully verified 14 propagated vulnerabilities.**

***Index Terms*—Vulnerability propagation; Proofs-of-Concept; taint analysis; symbolic execution.**

I. INTRODUCTION

Recent years have seen a considerable increase in the number and reuse of open-source software (OSS) [1]–[4]. Software developers benefit from this practice by reusing functionalities from reliable OSSs rather than re-inventing complicated wheels. However, as pieces of code are often reused by multiple parties, vulnerabilities discovered in one code propagate considerably to other software, resulting in a threat to the security of OSS ecosystems [5], [6].

One effective means of resolving this issue is to leverage vulnerable code clone detection techniques [6]–[8] that detect propagated vulnerable code clones in various software programs. Using these techniques, developers can detect vulnerabilities that have been propagated to their software and can further apply proper patches to vulnerabilities.

Despite their effectiveness, determining whether a vulnerable code clone can *actually* be triggered is mostly beyond the scope of existing vulnerable clone detection techniques. Indeed, a vulnerable code clone found in specific software does not always affect that software. This is primarily because the cloned vulnerable code (1) may not be called in the propagated software, (2) may be excluded during software build process, and further, (3) developers of propagated software may insert a patch of the vulnerable code that prevents the propagated vulnerability from being triggered.

* Heejo Lee is the corresponding author.

It is natural to fix (*e.g.*, patch) all detected vulnerabilities. However, this task not only incurs considerable costs and effort [9] but may also compromise the maintenance of the software project, for example, blindly removing vulnerable code can lead to syntax errors. Therefore, in addition to detecting propagated vulnerabilities, it is important to verify whether the propagated vulnerable code can still be triggered to achieve an efficient vulnerability management process such as prioritizing patching for more dangerous vulnerabilities.

To the best of our knowledge, none of the existing approaches are capable of verifying the triggerability of propagated vulnerable code effectively. On the one hand, several exploit primitive (*i.e.*, exploitable state) identification approaches exist that find triggerable vulnerabilities [10]–[15]. Yet, because these approaches focus only on detecting generic bugs or crashes in a software project, they not only consume considerable time in verifying *specific* (*i.e.*, propagated) vulnerabilities, they also can hardly determine the propagated vulnerability is not triggerable (see Section VI).

On the other hand, a Proof-of-Concept (PoC), a method for demonstrating the feasibility of a vulnerability, of the original vulnerability can be used to verify a propagated vulnerability. However, the execution path to reach the vulnerable code of the propagated software is typically different from that of the original vulnerable software, and thus, the PoC of the original vulnerability often fails to verify the propagated vulnerability.

Our approach. To overcome the shortcomings, we present a novel tool called **OCTOPOCS**, which is an implementation of an approach that uses a reformed PoC to verify whether a propagated vulnerable code can still be triggered in the propagated software. OCTOPOCS focuses on the C/C++ vulnerabilities with malformed file type PoCs (see Section II-A).

Let the original vulnerable software be S and the propagated software be T. To reform a PoC, OCTOPOCS (1) extracts the reusable part (called *crash primitive*) of the original PoC, which is used in the shared code area between S and T, (2) generates byte characters (called *guiding inputs*) that lead the execution flow of T from the entry point to the shared code area, and then (3) combines crash primitives and guiding inputs to create a new PoC.

We start with the idea that the part of a PoC used in the shared code between S and T can be reusable. We call this part the *crash primitive*. Note that the crash primitive may be used multiple times when triggering a single vulnerability. Thus, to extract the crash primitive from the original PoC,

OctoPoCs uses *context-aware taint analysis*, which is a type of taint analysis that recognizes the context in which and the number of times the program execution of S enters the shared code area to trigger the vulnerability. OctoPoCs extracts and stores the crash primitive of PoC used in the different contexts in a unit called a *bunch* (see Section III-A).

Thereafter, OctoPoCs generates guiding inputs that can lead to the execution flow of T from the entry point to the shared code area. In this process, naively using a symbolic execution often fails to generate guiding inputs as a result of the path explosion problem. Thus, OctoPoCs uses *directed symbolic execution* that can avoid path explosion. Specifically, OctoPoCs (1) analyzes the control-flow graph (CFG) of T, (2) finds the correct path from the entry point of T to the shared code area through backward path finding, and (3) uses the correct path to designate the direction of symbolic execution to generate guiding inputs (see Section III-B).

OctoPoCs then combines the crash primitive and guiding inputs to create a new PoC. In this process, each crash primitive is inserted into the new PoC at a different location (*i.e.*, offset) than that of the original PoC. To accomplish this, whenever the execution flow of T encounters the shared code area, OctoPoCs repeatedly inserts a corresponding *bunch* into a new PoC based on the file position indicator. For example, the bunch extracted when the first shared code area is reached in S is also inserted into the new PoC when the first shared code area is encountered in T (see Section III-C). Finally, OctoPoCs uses a reformed PoC to verify whether the propagated vulnerability is still triggered in T.

Evaluation. To evaluate OctoPoCs, we collected 15 real-world software pairs; each software pair shared the same known vulnerable code (where its PoC is publicly available) and received input as a file format (see Section V-A).

When OctoPoCs was applied using the collected software pairs, we confirmed that it successfully verified the propagated vulnerable code for 14 pairs. In particular, OctoPoCs discovered that propagated vulnerabilities could be triggered for the cases of nine software pairs, and for the remaining five cases, OctoPoCs verified that the vulnerabilities were not propagated (*i.e.*, the vulnerable code could not be triggered).

We then compared the effectiveness of OctoPoCs with that of existing approaches, AFLFast [16] and AFLGo [17]. We used three software pairs in which many changes were made in the original PoC during the process of PoC reform. Although AFLFast and AFLGo had been operating for more than 20 h, we confirmed that AFLFast was able to verify only one of the cases, and AFLGo failed to verify any of them, whereas OctoPoCs could verify the vulnerabilities for all three cases within 15 min (see Section V).

OctoPoCs further discovered three software programs in which the propagated vulnerability was triggered even in their latest versions: `libgdx`, `tjbench` in Mozilla mozjpeg, and `pdftops` in Xpdf (see Section V-B). We reported this information to all of the development teams. Libgdx and Xpdf teams immediately patched the vulnerability; specifically, the Xpdf team further assigned the new CVE ID for this vulnerability (CVE-2020-35376). Mozilla team responded that the vulnerability would be patched immediately.

This paper makes the following three contributions:

- We present a novel tool, OctoPoCs, that can effectively verify the triggerability of propagated vulnerable code using a reformed PoC. The source code of OctoPoCs is available at https://github.com/blbi/OctoPoCs/.
- Our key technical contribution is the PoC reforming technique, which extracts the crash primitive using context-aware taint analysis and generates guiding inputs using directed symbolic execution.
- We demonstrate the effectiveness of OctoPoCs using 15 real-world software programs; OctoPoCs successfully verified the propagated vulnerable code for 14 programs.

II. Approach of OctoPoCs

In this section, we introduce background information, key features of OctoPoCs, and motivating examples.

A. Background

Verification of propagated vulnerable code. To avoid confusion, we define the manner in which the propagated vulnerability used in this paper was verified. First, we assumed that the vulnerable code was propagated from the original vulnerable software S to the propagated software T. However, the presence of vulnerable codes does not always indicate that vulnerability can be triggered. Therefore, we checked whether the vulnerability could still be triggered in T. The result was that the vulnerability was either triggerable or not triggerable. This task was defined as the verification of propagated vulnerable code.

PoC. In software security, a PoC indicates a method for demonstrating the feasibility of a vulnerability. Various types of PoC exist, and we classified them into the following four categories [18]:

(1) Shell command type;
(2) Program type (*e.g.*, python script code);
(3) Malformed string type;
(4) Malformed file type (*e.g.*, malicious image file).

OctoPoCs verifies propagated vulnerabilities by reforming the original PoC and utilizing it. Notably, each type of PoC requires a different reforming approach. For example, in the cases of malformed file type PoCs, OctoPoCs must modify the byte characters within the original PoC. In this paper, we primarily focused on *malformed file type PoCs* based on our experimental observations. After investigating all reported Common Vulnerabilities and Exposures (CVE) from 2016 to 2019, we discovered 2,455 CVEs that include Bugzilla-report URLs as a reference. Of the total, 1,190 CVE vulnerabilities were reported along with a PoC, and we found that 823 PoCs (70%) were malicious file types, whereas all other types accounted for only 30%. Therefore, OctoPoCs targets the

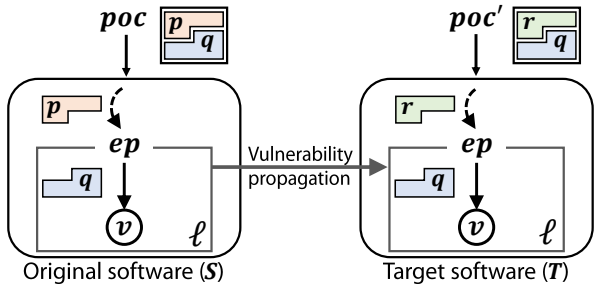

Fig. 1: Depiction of vulnerability propagation and the concept of PoC reforming.

PoC of the malicious file type, which accounts for the largest portion of all PoC types.

Taint analysis. Taint analysis is a technique that tracks an untrusted input in the software and determines which memory addresses and registers are affected [19]–[21]. The main idea of taint analysis is to continuously mark memory addresses and registers that belong to the data flow of untrusted input. With the help of taint analysis, we can track memory addresses and registers that can be controlled by a specific untrusted input.

Symbolic execution. Symbolic execution is a software analysis technique, which executes a software program with symbolic variables and identifies some constraints of the symbolic variables under each branch condition. In addition, symbolic execution produces formulas for the symbolic variables according to the constraints [22]–[24]. As a result, we can determine which values are needed to reach a specific location in the program by solving the formulas of symbolic variables.

B. Overview of OctoPoCs

Figure 1 presents the concepts of vulnerability propagation and PoC reform used in this paper, and Table I provides detailed descriptions of the defined notations. We use these notations throughout this paper.

Note that a file-type PoC is composed of byte characters. When poc is given as the input file to the software, the execution path of the software is determined by the byte characters contained in poc. If poc is capable of verifying v in S, the execution path of S should eventually reach v. Now, let us assume that the vulnerable code has propagated from S to T. At this time, the internal execution path of ℓ, which contains the actual vulnerable code, does not be changed as shown in Figure 1. By contrast, the execution path of T from its entry point to ep mostly differs from that of S.

The goal of OctoPoCs is to verify propagated vulnerable code can still be triggered in T using an automatically generated poc'. OctoPoCs consists of the following four phases: (1) extracting *crash primitives* from poc (q in Figure 1), (2) generating *guiding inputs* (r in Figure 1), (3) combining the crash primitives and guiding inputs to create a new PoC (poc'), and (4) verifying v in T using poc'. The detailed design of OctoPoCs is provided in Section III.

TABLE I: Defined notations and descriptions.

Notation	Description
S :	Original software where the vulnerability originated.
T :	Software from which the vulnerability has propagated.
v :	A known vulnerability.
ℓ :	A set of functions shared by both S and T.
q :	Crash primitive.
p, r :	Guiding inputs.
ep :	Entry point of ℓ. This ep is the function called first in ℓ in the process of triggering v.
poc :	An input file (PoC) that triggers v in S.
poc' :	An input file (PoC) that triggers v in T.

C. Motivating example

Triggered case. In 2017, a null pointer dereference vulnerability was discovered in the `mutool` binary of MuPDF software [25]. The vulnerable code originated in OpenJPEG and then propagated to MuPDF, which reused the OpenJPEG codebase. This vulnerability was triggered when a malicious JPEG2000 image file (*e.g.*, j2k file) was decoded. However, because the `mutool` binary can receive only a PDF file as input, even if the malicious j2k file had been given to the `mutool` binary, the vulnerability would not have manifested, as shown in Figure 2.

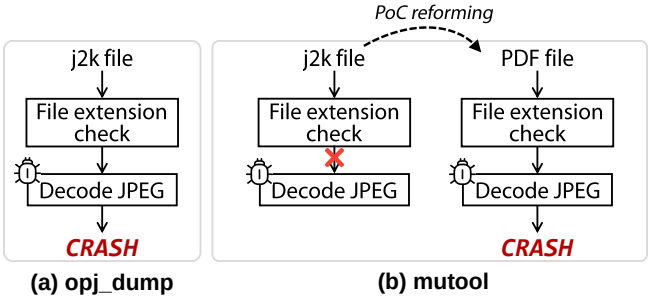

Fig. 2: Depiction of vulnerability verification in mutool through PoC reforming.

Based on this, OctoPoCs first generates a guiding input for the `mutool` binary by changing the malicious image file to a PDF file format. OctoPoCs then creates a new poc (poc') by combining the crash primitive of the original poc and the guiding input. It can then successfully verify the propagated vulnerability within the `mutool` binary.

Non triggered case. In 2016, a stack-based buffer overflow vulnerability was discovered in LibTIFF, which could cause a denial of service attack (CVE-2016-10095). This vulnerability originated in the `tiffsplit` binary in LibTIFF. The vulnerable code snippet is shown in Listing 1.

Listing 1: The vulnerable code snippet in tiffsplit.

```
1  //The vulnerability appears when "tag == 0x13d"
2  static int
3  _TIFFVGetField (TIFF* tif, uint32 tag, va_list ap)
4  {...
5      uint32 standard_tag = tag;
6      ...
7      switch (standard_tag) {
8          case TIFFTAG_SUBFILETYPE:
9              *va_arg(ap, uint32*) = td->td_subfiletype;
10             break;
11         case TIFFTAG_IMAGEWIDTH:
12             *va_arg(ap, uint32*) = td->td_imagewidth;
13             break;
14     ...}
```

Specifically, this vulnerability is triggered when the "tag" parameter is `0x13d`. We discovered that the *tiftoimage* function in the `opj_compress` binary of OpenJPEG contains the same vulnerable code. However, the *tiftoimage* function receives only hard-coded seven "tag" values as parameters except for the `0x13d` value that causes the actual vulnerability. Uncovering the fact that propagated vulnerable code cannot be triggered is crucial for prioritizing vulnerability patches. When OCTOPOCS tried to generate poc', it discovered that generating a guiding input was infeasible, indicating that the vulnerability could be triggered in the `opj_compress` binary.

Existing exploit primitive identification approaches are not designed to verify such propagated vulnerabilities. In particular, these approaches are not applicable to determine the fact that the vulnerabilities in propagated code could not be triggered. Furthermore, these approaches often require a huge amount of time to verify a triggerable propagated vulnerability. For example, we ran AFLFast [16] on the `mutool` binary for 20 h, but it failed to verify the aforementioned vulnerability [25] in the "triggered case" part.

III. Design of OctoPoCs

In this section, we describe the main design of OCTOPOCS.

Design assumption. We assume that OCTOPOCS, by its ability to leverage existing vulnerable clone detection techniques (*e.g.*, [6], [8]), already knows the S and T pairs whereby a vulnerable code has propagated from S to T. Given the source code of S and T, the methodology of the existing vulnerable clone detection technique allows us to detect whether a vulnerable code has propagated from S to T. In addition, we can also find ℓ, that is, the shared functions between S and T by using its mechanism, which is based on the code clone detection technique. Therefore, we assume that the initial inputs of OCTOPOCS are S, T, poc, and ℓ.

Design overview. Figure 3 illustrates the high-level workflow of OCTOPOCS. OCTOPOCS consists of the following four phases: (1) **P1** for extracting crash primitive, (2) **P2** for generating guiding input, (3) **P3** for combining crash primitive and guiding input to generate a new PoC, and (4) **P4** for verifying the propagated vulnerable code.

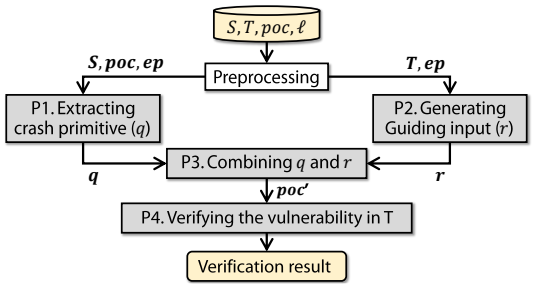

Fig. 3: High-level workflow of OCTOPOCS. OCTOPOCS consists of four phases (P1 to P4), and a final output is poc', which is used to verify propagated vulnerabilities.

In **P1**, for given S, ep, and poc, OCTOPOCS extracts crash primitives of poc (*i.e.*, q), which are the reusable parts of poc, by utilizing context-aware taint analysis:

$$q = P1(S, ep, poc)$$

In **P2**, for given T and ep, OCTOPOCS generates the guiding input r, which is a set of byte characters that lead to the program execution flow to ep, by using directed symbolic execution:

$$r = P2(T, ep)$$

In **P3**, for given q, r, and T, OCTOPOCS combines q and r to generates poc':

$$poc' = P3(T, q, r)$$

Finally, in **P4**, OCTOPOCS verifies the propagated vulnerability in T using poc'.

Preprocessing. Before generating poc', OCTOPOCS first identifies ep by using `backtrace` function [26]. This function enables OCTOPOCS to check which functions are called (*i.e.*, callstack) when the vulnerability v in S is triggered by poc. Among the called functions, we define a function as ep when it satisfies the following two conditions: (1) it should belong to ℓ and (2) it should be the bottommost function included in the callstack when v is triggered. In other words, ep is the first function to be called in ℓ. Subsequently, the discovered ep is utilized to generate poc'.

A. Extracting crash primitive phase (P1)

Phase overview. This phase starts with three inputs: S, poc, and ep. The goal of P1 is to extract the crash primitive from poc which is the reusable part when reforming poc'. We previously defined the shared parts between S and T as ℓ (see Figure 1). In other words, the problem of extracting the crash primitive is converted into a problem of extracting the specific bytes of poc used in ℓ.

Specifying the memory area of interest. To extract crash primitives, OCTOPOCS must monitor which bytes of poc are used in each execution flow of S when S is executed with poc. To this end, OCTOPOCS first specifies the memory area to which the bytes of poc are uploaded. Note that input file data are stored in memory primarily by using two methods: using

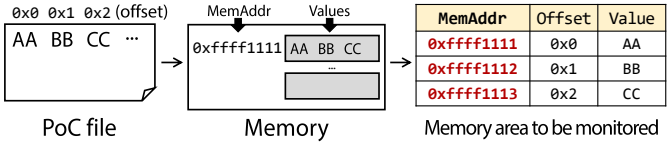

Fig. 4: Designation flow for the memory area to be monitored.

a *file-read function* and using a *memory-mapping function*. Hence, OCTOPOCS hooks all file-read and memory-mapping functions in S, including *poc* as the parameter. This can be confirmed through the file descriptor [27]. OCTOPOCS designates all memory addresses where *poc* bytes are uploaded as the monitoring area. Each memory address value is stored along with the corresponding *offset* of the file byte that affected the memory address. The example flow for specifying the memory area to be monitored is shown in Figure 4.

Monitoring the memory address. Once OCTOPOCS specifies the memory area where the bytes of *poc* are uploaded (called the specified memory area), the next step is to trace this memory area while executing S. When extracting the bytes of *poc* used in ℓ, it seems sufficient to consider the program execution only after encountering *ep*. However, some bytes in *poc* may be read and stored before entering ℓ and then *indirectly* used in ℓ. Thus, we decided to trace the specified memory area from the entry point of S.

Context-aware taint analysis. Another major consideration is that *ep* can be called multiple times. To address this issue, OCTOPOCS uses *context-aware taint analysis*. This method understands the context in which the software is executing. In particular, OCTOPOCS uses taint analysis that recognizes (1) the parameters of *ep*, (2) the number of times the program execution encounters *ep*, and (3) the byte characters of *poc* that are used each time it goes into ℓ. The flow of monitoring the specified memory area is given as follows:

- **P1.1** During the execution of S, OCTOPOCS checks whether the specified memory area is referenced by any memory read and write operations.
- **P1.2** Any memory addresses and registers affected by the specified memory area are marked as new trace areas. For all newly marked memory addresses while tracing, we call them *candidate addresses*, indicating the parts that can be indirectly used in ℓ. OCTOPOCS repeats **P1.1** and **P1.2** until the program execution encounters *ep*.
- **P1.3** If an access to the specified memory area occurs after *ep* is called (*i.e.*, the program execution goes inside ℓ), all file bytes of *poc* that affected the accessed memory address are marked as crash primitives. In addition, even when an access to the candidate memory addresses occurs, the file bytes of *poc* that affected the accessed memory address are also marked as crash primitives.

Once OCTOPOCS performs **P1.3**, OCTOPOCS groups the marked crash primitives (*i.e.*, byte characters used in ℓ at the same sequence) into a *bunch*; multiple bunches are generated if the execution path of S enters ℓ more than one time when triggering v. Each bunch is stored along with the number of encounters with *ep* (sequential value). For example, when the execution flow of S encounters *ep* for the second time, the bunch is saved with a value of 2. Later, OCTOPOCS generates *poc*′ through a combined step that considers all of these bunches.

Finally, when a crash occurs in S, the crash primitive extraction phase ends and all bunches collected thus far (*i.e.*, crash primitives) are passed to the next step. The high-level algorithm of this phase is explained in Algorithm 1.

Algorithm 1: Algorithm for extracting crash primitives.

```
Input: S, poc, ep
Output: q // Crash primitives of poc.
1  procedure ExtractingCP(S, poc, ep)
2      q ← ∅
3      Run(S, poc)
4      poc_mem ← MemoryHooking(S)
       // Memory address set of poc bytes.
5      tainted ← poc_mem
       // Tainted objects (e.g., memory addresses and registers).
6      while ¬ CRASH do
7          read_obj, write_obj ← GetCurrentAsm()
8          if read_obj ∈ tainted then
9              tainted.add(write_obj)
10         else if write_obj ∈ tainted then
11             tainted.del(write_obj)
12         if Enter(ep) then
13             epCount ← GetCountEp()
               // Measures #times ep has been encountered.
14             value ← GetPoCValues(read_obj)
               // Get the file byte values corresponding to read_obj.
15             q[epCount].add(value)
16     return q
```

B. Generating guiding inputs phase (P2)

Phase overview. This phase starts with two inputs, T and *ep*. The goal of P2 is to generate guiding inputs to lead the execution flow of T from the entry point to *ep*. To this end, OCTOPOCS decides to use symbolic execution. However, when generating guiding inputs, finding values that lead execution flow to *ep* with naive symbolic execution often causes a path explosion (especially when T has complex branches). To avoid this issue, we use *directed* symbolic execution. Specifically, OCTOPOCS first determines the correct path from the entry point of T to *ep*, and then detects the constraints to reach *ep* using the correct path information.

Backward path finding. OCTOPOCS first generates the control-flow graph (CFG) of T to determine the correct path from the entry point of T to *ep*. This generated CFG provides information about the possible execution flows of T. Because many branches exist in T, simply tracing the program execution flows from the entry point of T to *ep* results in considerable computational cost. Because we know

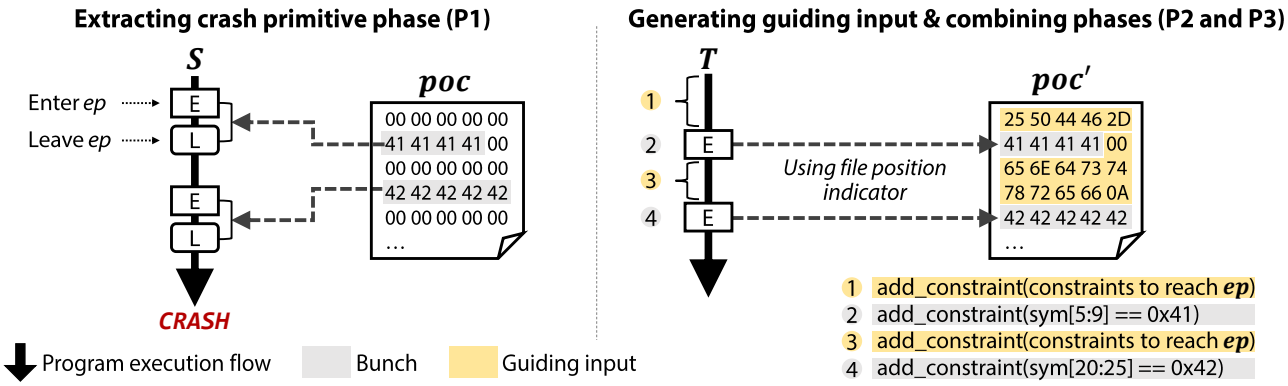

Fig. 5: Illustration for generating poc'.

the location of ep, OctoPoCs detects the correct path (from the entry of T to ep) by tracing backwards (from ep to the entry of T). This approach enables OctoPoCs to reduce considerably the number of paths to be tracked and the number of computing resources.

Directed symbolic execution. OctoPoCs now knows the correct path from the entry point of T to ep. OctoPoCs next uses symbolic execution to extract constraints that satisfy the correct path. Initially, the input file given to T is a file in which all bytes are designated as symbols (this input symbol file becomes poc' after going through the process up to **P3**).

OctoPoCs collects constraints in the following manner:

P2.1 Whenever a branch is encountered during executing T, OctoPoCs points in the proper direction using the correct path information determined during backward path finding. Because the program execution does not follow unnecessary paths, OctoPoCs can avoid path explosion and optimize poc' to be generated.

P2.2 OctoPoCs collects all the constraints obtained through symbolic execution until the program execution of T encounters ep. OctoPoCs will solve all the constraints at the combining phase (*i.e.*, P3, see Section III-C).

We define four types of states that the execution flow of T can encounter during symbolic execution as follows: (1) *active*, (2) *loop*, (3) *loop-dead*, and (4) *program-dead*.

An active state is a normal state, and OctoPoCs does not have to do anything additional. However, OctoPoCs must carefully address the other three types of states when collecting constraints. A loop state means that the state of symbolic execution falls into a loop. In the loop state, determining the correct number of iterations to exit the loop state and finally reach ep is not easy. Thus, OctoPoCs decides to handle the state by increasing the number of iterations from one to θ (*i.e.*, the maximum number of iterations, see Section IV-B) and repeating the loop state until the program execution does not reach the loop-dead state. Here, a loop-dead state refers to a state in which a concrete value that satisfies all the collected constraints does not exist when exiting the loop using the selected iteration.

Lastly, a program-dead state indicates that the program execution falls into a dead state after exiting a loop with θ iterations or before encountering any loop states. If the program execution encounters a program-dead state, OctoPoCs determines that ℓ is not reachable, indicating that the vulnerability cannot be triggered in T. In contrast, if the program execution flow successfully reaches ep, the constraints to reach from the entry point of T to ep can be collected.

C. Combining phase (P3)

This phase starts with three inputs: q, r, and T. The goal of P3 is to combine the obtained q and r to generate poc'. It should be noted that P3 is not a completely separate phase from P2; in fact, P3 proceeds whenever the execution flow of T encounters ep in P2. Initially, an input file filled with symbols was given as the input of T, and then constraints for guiding inputs were collected in P2. In this phase, constraints for crash primitives (described in **P3.1**) are applied to symbols in the corresponding part of the input symbol file. Thereafter, OctoPoCs solves all the collected constraints and replace the original symbols with concrete values. Finally, the input symbol file with concrete values becomes poc'.

This task may seem easy at first glance, but it is error-prone primarily for the following two reasons: (1) the size of guiding inputs for T is often differ from that of S, and (2) ep may be called multiple times in T. Because of the first reason, placing q to poc' using the same offset of q in poc results in an error. In addition, q is a set of byte characters used after the execution flow encounters ep; since ep can be encountered multiple times during the execution of T, OctoPoCs needs to insert q in the proper position of poc' for every context.

To address such issues, whenever the execution flow of T encounters ep, OctoPoCs checks the corresponding file position indicator of the input symbol file and confirm the proper position (*i.e.*, offset) of q in poc'; note that OctoPoCs executes ep in T with the same parameters as those used in S. Furthermore, OctoPoCs extracts q into bunches according to the sequence for encountering ep in P1. Therefore, whenever the execution flow of T encounters ep, each corresponding bunch is placed so that it can be used in the correct order.

The detailed flow of the combining phase is as follows:

P3.1 Whenever the execution flow of T encounters ep, the corresponding bunch (according to the sequential number extracted in P1) is combined with r. In this process, to avoid constraint conflict between q and r, file bytes of q are expressed in constraint form. For example, when OCTOPOCS places a bunch (`0x41414141`) in poc' from offset 5 to 8, the bunch is stored in the form of constraint as "`sym[5:9] == 0x41`" (see Figure 5). The offset to place q is determined by the file position indicator when entering ℓ. The file position indicator is a pointer to the current byte to read in the file.

P3.2 If the execution flow of T enters ℓ multiple times, OCTOPOCS repeats the previous step (**P3.1**) each time when the execution flow enters ℓ.

P3.3 After the final encounter of ep, OCTOPOCS terminates the execution of T and solves the constraints gathered thus far. When concrete values that satisfy all constraints are determined, the symbolic values of the input symbol file are replaced with these concrete values. The file filled with these concrete values becomes poc'. If concrete values satisfying all constraints are not generated, the vulnerability cannot be triggered in T.

Figure 5 describes the sample poc' generation process. In this example, ep is called twice in P1, and each time `0x41414141` and `0x4242424242` bunches are stored. Then, in P2 and P3, OCTOPOCS obtains constraints for guiding input and every time the execution flow of T encounters ep, constraints for each bunches are added to the corresponding symbols in the input symbol file. Finally, after solving all the added constraints poc' is generated. Algorithm 2 shows the high-level workflow of P2 and P3.

Algorithm 2: Algorithm for generating guiding inputs and combining phases.

```
Input: T, ep, q
Output: new_poc // poc'
1  procedure GeneratingGI(T, ep, q)
2      next_direction ← initial_direction
3      epCount ← 0
4      path2ep ← FindPath(CFG of T)
5      constraint ← Ø
6      while CanExecute() do
7          branch = Execute(next_direction)
8          constraint.add(GetConstraint(next_direction))
9          if Enter(ep) then
10             pos = GetCurrentPosition()
11             LocatingCP(pos, q[epCount], constraint)
               // Add constraints of q[epCount]
12             epCount += 1
13             if epCount == q.len() then
14                 break
15             ExploreWhileEp()
16         next_direction = GetPath(branch, path2ep)
17     new_poc = Eval(constraint)
18     return new_poc
```

D. Verifying the propagated vulnerability phase (P4)

Finally, if poc' has been generated, OCTOPOCS verifies that the propagated v is still triggerable in T using poc'. To do this, OCTOPOCS executes T with the generated poc', and checks whether poc' works on T (*e.g.*, cause a crash).

Cases of successfully verifying vulnerability. Three possible cases of successful verification exist:

(i) If OCTOPOCS causes a crash in T using poc';
(ii) When ep is not called in T;
(iii) When facing a program-dead state.

In particular, case (i) indicates that the propagated vulnerability can still be triggered in T. This vulnerability poses a real threat; thus, immediate patching is required. Cases (ii) and (iii) are the results of verification in which the propagated vulnerable code cannot be called or triggered in T, indicating that T is currently not threatened by the propagated vulnerable code (even though it must be patched in the end).

Cases of failing to verify the vulnerability. OCTOPOCS may not be able to generate poc' for several reasons other than the those described in cases (ii) and (iii). The main cause of failure is the loop state within T. OCTOPOCS runs the loop state repeatedly up to θ until the execution flow escapes the loop-dead state. However, repeating the loop beyond what our directed symbolic execution can cover is necessary. Although this case was not found in the current evaluation, it may occur later. Thus, improving OCTOPOCS so that it can efficiently handle loops is essential. We leave this as future work.

IV. Implementation of OctoPoCs

OCTOPOCS comprises two modules: (1) taint analysis engine module (implemented in 2,400 lines of C++ code) and (2) symbolic execution engine module (implemented in 500 lines of Python code). We used Intel PIN [28] and angr [29] to support each engine, respectively. Although the methodology used in OCTOPOCS is not restricted to a particular language, we implemented OCTOPOCS targeting C/C++ software; this is because, source code reuse is prevalent in C/C++ software [6].

A. Taint analysis engine module

The taint analysis engine module is designed for extracting crash primitives from the original PoC. We implemented this module using the dynamic binary instrumentation (DBI) technique, a binary analysis technique that inserts the commands we require. This allows us to designate the specific behavior of the program when each of the instructions or functions are executed. In particular, we use PIN [28], a DBI tool developed by Intel, as it provides various APIs that enable to check context information such as registers. PIN emulates the target software program with our taint analysis engine module.

In addition, there are two types of file-read functions: *system call* and *library call*. Because frequent use of system

calls makes the software program unable to utilize resources efficiently, developers often make use of library calls. Hence, OCTOPOCS considers both system call (*e.g.*, *__NR_read*) and library call (*e.g.*, *fread*) when hooking a file read function.

Furthermore, software S processes *poc* at the byte character-level. Therefore, we also handle the tainting at the byte character-level. All byte characters in *poc* are checked if they are tainted or not individually. Lastly, OCTOPOCS controls both 64bit and 32bit registers when performing taint analysis.

B. Symbolic execution engine module

The symbolic execution engine module is designed to generate guiding inputs and combines crash primitives and the generated guiding inputs. We use angr [29] to support this module, a powerful and widely used framework for symbolic execution. Angr provides many APIs that can analyze software with symbolic values, along with the solver engine that can solve the constraints of symbolic variables. In addition, angr provides a function to generate CFG, which is used to find the correct path (see Section III-B). We use angr especially when setting symbolic bytes in a file and executing a program with the symbolic file. We further calculate the constraints of symbolic bytes in the file using the solver engine of angr to obtain concrete values of the bytes in the file.

In addition, we previously set θ, the maximum number of iterations for existing a loop state when generating guiding inputs (see Section III-B). When we evaluate OCTOPOCS, we set θ as 120. This value was obtained from our experimental results; in most cases, we confirmed that OCTOPOCS could exit a loop state before conducting 120 iterations.

Lastly, two types of CFG exist: static and dynamic CFG. A static CFG is generated only considering the call and jump blocks (*e.g.*, `jmp`). Although a static CFG can be quickly generated with highly accurate block information, it cannot contain the indirect call edge that appears only when a program is running. In contrast, a dynamic CFG is generated with symbolic execution; transition appears only in execution time. Although the cost required to create a CFG is high, it is possible to cover even functions that appear only during program running, we determine to use the dynamic CFG mainly; however, we have the option of using a static CFG.

V. EVALUATION

We next evaluated OCTOPOCS. The main objective was to demonstrate how OCTOPOCS effectively verified a propagated vulnerability. We introduce the collected dataset for evaluating OCTOPOCS in Section V-A. In Section V-B, we investigate the vulnerability verification results of OCTOPOCS. Section V-C explains the effectiveness of the methodology used in OCTOPOCS (*i.e.*, context-aware taint analysis and directed symbolic execution). Finally, we demonstrate the effectiveness of OCTOPOCS by comparing OCTOPOCS with the existing approaches, AFLFast [16] and AFLGo [17], in Section V-D. We evaluated OCTOPOCS on a machine with Ubuntu 16.04, Intel Core i7-7700 CPU @ 3.60GHz, and 32GB RAM.

A. Dataset collection

To collect the dataset, we first viewed the top 500 C/C++ software as ranked by the number of reported CVE vulnerabilities in the National Vulnerability Database (NVD). Among them, software programs suitable for evaluation were chosen based on the criteria that (1) they receive a file format as input (as OCTOPOCS targets malformed file type PoCs) and (2) the size of their binaries should be small enough to work on angr [29] used for the implementation. There were only 17 software programs that satisfied the criteria, and we considered them to be the original vulnerable software S. Next, we used the methodology and dataset of VUDDY [6] (the scalable vulnerable code clone detection technique) to detect T. We searched for software that shares a vulnerability code with S among the software dataset of VUDDY. Among the searched software, we selected a software program as T when it satisfies the two criteria mentioned previously. Note that we only considered cases in which a PoC of the vulnerability was publicly available on a website such as Bugzilla or GitHub.

Finally, 14 pairs of S and T were collected as the dataset. In general, multiple binaries exist in a software project. We consider only a binary that was directly related to a vulnerability by referring to vulnerability-related information such as the NVD vulnerability description. In addition, we decided to add one artificial test case as it can show some interesting results. The collected binaries, ranging from 2,000 to 557,000 lines of code, were from diverse domains: namely, multimedia (*e.g.*, ffmpeg), PDF (*e.g.*, pdfalto and ghostscript), and image (*e.g.*, gif2png and opj_dump) related binaries. The number of collected binaries is as many as that collected in related approaches; *e.g.*, AFLFast utilized less than 10 real-world binaries, and AFLGo utilized less than 20 binaries.

B. Propagated vulnerability verification

Types of verification results. We first classifed the vulnerability verification results of OCTOPOCS into four types:

- *Type-I:* Cases in which the guiding inputs of *poc* and *poc′* were the same, and in which the propagated vulnerabilities were triggered in T.
- *Type-II:* Cases in which the generated guiding input for *poc′* was different from that of *poc*, and in which the propagated vulnerabilities were triggered in T.
- *Type-III:* Cases in which OCTOPOCS confirmed that the vulnerabilities were not propagated to T (*i.e.*, the propagated vulnerability could not be exploitable).
- *Failure:* Cases in which OCTOPOCS failed to verify the propagated vulnerable source code.

We also manually checked the results for *Type-III* cases to determine whether they were correct (*e.g.*, whether the vulnerability is not triggerable).

Results. OCTOPOCS succeeded in verifying the propagated vulnerabilities in 14 of the 15 datasets. Detailed verification results for OCTOPOCS are listed in Table II.

TABLE II: Vulnerability verification results of OCTOPOCS. Idx means the index number of each result, S indicates the original binary where the known-vulnerability originated, T represents the target binary that has a vulnerable clone of S, poc' denotes whether OCTOPOCS generated poc' for the vulnerability, and the verification column indicates whether OCTOPOCS can verify the vulnerability in T.

Type	Idx	S		T		Vulnerability		poc'	Verification
		Name	Version	Name	Version	ID	Type†		
Type-I	1	JPEG-compressor	N/A	libgdx	1.9.10	CVE-2017-0700	No-CWE	O	O
	2	JPEG-compressor	N/A	zxing	@0a32109	CVE-2017-0700	No-CWE	O	O
	3	pdftops (Poppler)	0.59	pdftops (Xpdf)	4.02	CVE-2017-18267	CWE-835	O	O
	4	avconv	12.3	ffmpeg	1.0	CVE-2018-11102	CWE-119	O	O
	5	tjbench (libjpeg-turbo)	2.0.1	tjbench (mozjpeg)	@0xbbb7550	CVE-2018-20330	CWE-190	O	O
	6	pdfalto	0.2	pdfinfo (Xpdf)	4.0.0	CVE-2019-9878	CWE-119	O	O
Type-II	7	ghostscript	9.26	opj_dump	2.1.1	ghostscript-BZ697463	No-CWE	O	O
	8	opj_dump	2.1.1	MuPDF	1.9	ghostscript-BZ697463	No-CWE	O	O
	9	gif2png	2.5.8	gif2png (artificial)	N/A	CVE-2011-2896	CWE-119	O	O
Type-III	10	tiffsplit	4.0.6	opj_compress	2.3.1	CVE-2016-10095	CWE-119	X	O
	11	tiffsplit	4.0.6	libsdl2	2.0.12	CVE-2016-10095	CWE-119	X	O
	12	tiffsplit	4.0.6	libgdiplus	6.0.5	CVE-2016-10095	CWE-119	X	O
	13	ghostscript	9.26	opj_dump	2.2.0	ghostscript-BZ697463	No-CWE	X	O
	14	pdfalto	0.2	pdftops (Xpdf)	4.1.1	CVE-2019-9878	CWE-119	X	O
Failure	15	pdf2htmlEX	0.14.6	pdfinfo (Poppler)	0.41.0	CVE-2018-21009	CWE-190	X	X

† CWE-119: buffer overflow, CWE-190: integer overflow, CWE-835: infinite loop

A total of six *Type-I* cases existed. In these cases, we found that poc' was often more optimized than poc because it did not contain unnecessary bytes when a vulnerability was triggered. For *Type-I* cases, poc itself could be used to verify the propagated vulnerability. However, it proved significant that OCTOPOCS effectively verified the propagated vulnerability after generating poc'.

In addition, three *Type-II* cases were observed in the results. Idx-7 and 8 as listed in Table II were those cases in which OCTOPOCS could verify the propagated vulnerability using the reformed PoC obtained by changing the header part of the original JPEG file into PDF file format and *vice versa* (as described in Section II-C). Idx-9 was an artificial test case. The original gif2png binary (v2.5.8) contains the *ReadImage* function, which includes the vulnerable code that can cause a heap-based buffer overflow (CVE-2011-2896). The PoC of the vulnerability was disclosed as a GIF file format, which should contain version information (*e.g.*, GIF87a) in its header section. However, we found that invalid GIF version was recorded in the disclosed PoC that we collected, and gif2png does not care about invalid version information during the processing of image data. Thus, we modified the version checking of gif2png more strictly: if the GIF version information of the input image file was invalid, the gif2png binary exited immediately. Consequently, OCTOPOCS successfully generated a new PoC that had valid GIF version information in the header section. The generated PoC worked well on the modified gif2png.

Furthermore, we found five *Type-III* cases in which OCTOPOCS failed to generate poc' but confirmed that the vulnerability was not triggerable (*i.e.*, the propagated vulnerable code could not be exploitable). In the cases of Idx-10 to Idx-12 as listed in Table II, the vulnerability was not triggered for the same reason as described in Section II-C: although the *_TIFFVGetField* function (vulnerable function) was cloned in T, OCTOPOCS confirmed that it was being reused in an environment in which the tag value used in causing the vulnerability could not be delivered. For the remaining cases (Idx-13 and 14), the results of the testing were for whether the vulnerability could be triggered in *patched* versions of T that succeeded in triggering the vulnerability at Idx-7 and Idx-6, respectively. As expected, we verified that the vulnerability was not triggered after a patch code was inserted in T.

Finally, there was one case in which OCTOPOCS failed to verify the propagated vulnerability, namely, Idx-15, as listed in Table II. We determined that this failure was not due to a problem with the OCTOPOCS methodology, Instead, it was caused by angr as used in the implementation, where angr did not correctly create the CFG of pdfinfo (due to a bug in its codebase). We reported this bug to the angr team (April 2020) and currently await a fix. If this bug is resolved, we expect that the case can also be verified by OCTOPOCS.

Interestingly, three results were obtained in which the propagated vulnerability was triggered in the latest version of T: libgdx, tjbench in Mozilla mozjpeg, and pdftops in Xpdf (Idx-1, 3, and 5). In 2018, a dangerous integer overflow vulnerability was discovered in tjbench of libjpeg-turbo. Although the libjpeg-turbo team immediately patched the vulnerability (Nov. 2018), we confirmed that the vulnerability still remained in the latest version of Mozilla mozjpeg, which reuses code from libjpeg-turbo (Jan. 2020). Similarly, the other two results were cases in which the propagated vulnerability was not patched to the latest version of each T. We reported these results to the corresponding development teams with requests for patches. The libgdx and Xpdf teams immediately patched the vulnerability through our report; Xpdf team further assigned the new CVE ID (CVE-2020-35376) for this vulnerability. The Mozilla team responded that they would analyze the vulnerability and patch it as soon as possible.

TABLE III: Effectiveness of context-aware taint analysis.

Idx	S	T	Taint analysis†	Context-aware taint analysis
1	JPEG-compressor	libgdx	O	O
2	JPEG-compressor	zxing	O	O
3	pdftops (Poppler)	pdftops (Xpdf)	X	O
4	avconv	ffmpeg	X	O
5	tjbench (libjpeg-turbo)	tjbench (mozjpeg)	O	O
6	pdfalto	pdfinfo (Xpdf)	O	O
7	opj_dump	ghostscript	O	O
8	opj_dump	MuPDF	O	O
9	gif2png	gif2png (artificial)	X	O

†: taint analysis without context information.

C. *Effectiveness of* OCTOPOCS *methodology*

Effectiveness of context-aware taint analysis. We first evaluated the effectiveness of context-aware taint analysis used in extracting crash primitives (see Section III-A). We confirmed that the taint analysis technique without context information failed to generate poc' in three of nine datasets, whereas context-aware taint analysis successfully generated poc' for all cases, as shown in Table III. This was because, when we extracted crash primitives from a PoC without context information, even when ep was called multiple times during the execution of S, all the extracted crash primitives were located in poc' at once. Consequently, poc' generated in this manner was more likely not to work in T.

Effectiveness of directed symbolic execution. We then evaluated the effectiveness of directed symbolic execution, which is used to generate guiding inputs (see Section III-B). To see how effectively guiding inputs are generated, we ran the experiment with only *Type-II* with large variations in the guiding input. *Type-I* and *Type-III* cases were excluded because *Type-I* cases have little difference in guiding input between poc and poc', and OCTOPOCS coud not generate poc' for the *Type-III* cases.

We measured the elapsed time and resources (*i.e.*, memory) consumed in reaching ep for each naive symbolic execution (offered by angr) and directed symbolic execution. Note that the directed symbolic execution proceeded with the correct path information to the vulnerable point (ep), but the naive symbolic execution proceeded with only an address of the vulnerable location. The results are shown in Table IV. The elapsed time and used memory differed depending on the complexity of T. Nevertheless, we confirmed that two of the three evaluation datasets with naive symbolic execution could not induce the execution of T until reaching ep due to memory errors (*i.e.*, path explosion problem). Our experimental results showed that the directed symbolic execution used by OCTOPOCS could successfully avoid the path explosion problem and generate guiding inputs in a reasonable time.

D. *Comparison with existing approaches*

Several existing approaches exist. However, most target only a specific type of vulnerability (*e.g.*, authentication bypass) or a specific environment (*e.g.*, kernel-related vulnerability); *e.g.*, Driller [11] was excluded from comparison because it only targets DARPA Experimental Cyber Research Evaluation Environment binaries. Thus, we selected AFLFast [16] (*i.e.*, coverage-based fuzzer) and AFLGo [17] (*i.e.*, directed fuzzer). We used the three cases listed in Table IV and compared the effectiveness of OCTOPOCS with AFLFast and AFLGo in terms of vulnerability verification. We ran AFLFast and AFLGo for 20 h in the same environment as that of OCTOPOCS.

TABLE IV: Effectiveness of directed symbolic execution.

S	T	SE†		D-SE‡	
		Time(s)	RAM(MB)	Time(s)	RAM(MB)
ghostscript	opj_dump	3.49	461	2.87	413
opj_dump	MuPDF	N/A	*MemError	74.56	937
gif2png	gif2png (arti.)	N/A	MemError	532.28	1,802

†: symbolic execution, ‡: directed symbolic execution, ∗: memory error.

TABLE V: Elapsed time for verifying the propagated vulnerability in T in AFLFast, AFLGo, and OCTOPOCS.

S	T	AFLFast*	AFLGo*	OCTOPOCS
		Elapsed time (s)		
ghostscript	opj_dump	N/A	N/A	9.67
opj_dump	MuPDF	N/A	Error†	75.4
gif2png	gif2png (arti.)	201	N/A	558.46

∗: running 20 h, †: cannot executed due to the tool error.

The comparison results are shown in Table V. Notably, OCTOPOCS required less than 15 min to verify the propagated vulnerabilities in the selected three binaries.

Although AFLFast verified the propagated vulnerability of gif2png in 201 s, for the other two cases, it could not verify the propagated vulnerabilities even after running 20 h. Existing fuzzers are greatly affected by the size of the testing binary. This is why AFLFast showed higher efficiency in gif2png. However, the other two binaries are large in size, thus AFLFast failed to verify the propagated vulnerabilities. By contrast, the efficiency of OCTOPOCS is more dependent on the size of the outer space of ℓ, because OCTOPOCS utilizes the concept of crash primitives. Even if the size of the binary is large, if the parts other than ℓ in binary are small, OCTOPOCS can verify the vulnerability in a reasonable time.

Similarly, AFLGo could not verify the propagated vulnerabilities within 20 h. Even though AFLGo uses the vulnerability location information, the input value to reach the vulnerable location in AFLGo was randomly generated, whereas OCTOPOCS uses the crash primitive extracted from the original PoC. There is a clear difference in performance and efficiency between using the known correct answers and generating the correct answer at random. Therefore, from the perspective of verifying the propagated vulnerability, OCTOPOCS is more efficient than AFLGo. For MuPDF case, due to a tool error, we could not test it with AFLGo. We reported this bug and are currently in discussion with the AFLGo team.

In conclusion, we confirmed that OCTOPOCS showed better performance than the existing fuzzers in terms of verifying the propagated vulnerabilities.

VI. Related work

Vulnerable code clone detection. There are a number of approaches attempted to detect vulnerable code clones [6]–[8], [30], [31]. Their main concern is to detect vulnerable code clones among software projects, and they are not interested in determining whether the detected vulnerable code clones are still triggerable after their propagation. Therefore, it is out of the scope of these studies to verify the triggerability of the propagated vulnerabilities, which we attempt to solve.

Exploit primitive identification. Several existing exploit primitive identification approaches exist that finds an exploitable state in software [10]–[15].

Brumley *et al*. [10] attempted to identified exploit primitives by finding differences between program P and its patched version P′. To find deeper bugs, Stephens *et al*. [11] used both fuzzing and selective concolic execution techniques to complement the shortcomings of each of them and combined only the advantages. However, their datasets are limited to applications from CGC (as mentioned in Section V-D), making the results questionable when applying their tool to real-world software. Lu *et al*. [12] used deterministic stack spraying method and exhaustive memory spraying technique to identify exploit primitive about use-before-initialization vulnerabilities in the Linux Kernel. Wu *et al*. [13] identified exploit primitives of use-after-free vulnerabilities in kernel with their tool, FUZE. Yun *et al*. [15] presented an automatic tool, ARCHEAP, to find the unexplored heap exploitation primitives. ARCHEAP works similar to fuzzing with common designs of allocators to find heap exploitation primitives. In addition, Wu *et al*. [14] proposed an exploit primitive evaluation framework, KEPLER, that generates exploit with "single-shot" exploitation chain. KEPLER gets a control-flow hijacking primitive as input and only focuses on making an exploit from the exploit primitive. However, this work focuses on the exploit primitive evaluation, the step after the exploit primitive identification.

Therefore, all of these approaches are not suitable to effectively perform verification for a specific propagated vulnerability, which we consider and attempt to resolve in this paper, without any restrictions in the type of vulnerabilities.

VII. Discussion

Malformed file type PoC. OctoPoCs only targets malformed file type PoC. Although we have confirmed that they account for a large portion of the reported PoCs that we observed (70%, see Section II-A), other types of PoCs still have to be considered. The concept proposed by OctoPoCs, using the reusable part of the original PoC (crash primitives) and efficiently generating the part to be added to a new PoC (guiding inputs) will be able to operate in other types of PoCs. If this becomes an issue, we will study the method for reforming other types of PoCs.

Practical usage. Assume that a developer has confirmed that several pieces of propagated vulnerable code exist in their software. At this point, they can use OctoPoCs to determine which vulnerabilities need to be patched more urgently (*i.e*., they can prioritize vulnerability patches). In addition, security analysts who are interested in vulnerability analysis can also use OctoPoCs to increase the security of the OSS ecosystem. For example, when they detect a propagated vulnerable code in a software program, they can easily check whether the vulnerability can still be triggered, and then recommend a patch to the corresponding software security team.

Code-fragments level vulnerability propagation. If the reused code area of S is minimal, *e.g*., only a few functions were reused, the crash primitives are minimized, so the performance of OctoPoCs can be comparable to that of other related approaches. However, this is an infrequent case, and the propagation of vulnerabilities mostly takes place in units of a set of numerous functions. The more functions were propagated, the more crash primitives were extracted, which can improve the performance of OctoPoCs.

Limitations. In our experiment, we used 15 vulnerable software pairs selected by clear criteria, but they may not be representative to demonstrate the effectiveness of OctoPoCs. Furthermore, when OctoPoCs attempts to exit a loop-state in T, OctoPoCs runs the loop state repeatedly up to θ until the execution flow of T exits the loop-dead state. However, certain vulnerabilities may be triggered only when iterating a specific loop statement more than θ times; currently, the method of OctoPoCs cannot be used to verify these kinds of vulnerabilities. Finally, we use angr when implementing OctoPoCs, which prevents vulnerability verification for binaries where angr does not work.

VIII. Conclusion

As code-reuse becomes prevalent in software development, addressing the propagation of vulnerability has emerged as an important concern. In response, we presented OctoPoCs for verifying whether a vulnerability in propagated vulnerable code can still be triggered by using the reformed PoC. The context-aware taint analysis and directed symbolic execution make OctoPoCs to effectively reform PoC and verify propagated vulnerabilities. Equipped with vulnerability verification results from OctoPoCs, we expect that developers can patch more dangerous vulnerabilities faster; in the end, OctoPoCs can contribute to making a more secure software ecosystem.

Acknowledgment

We appreciate our shepherd and the anonymous reviewers for their valuable comments to improve the quality of the paper. We are also grateful to Seungmok Kim and Geonwoo Lee for helping us analyze the experimental results manually. This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2019-0-01697 Development of Automated Vulnerability Discovery Technologies for Blockchain Platform Security and No.2021-0-01819 ICT Creative Consilience program).

REFERENCES

[1] S. Koch, "Evolution of open source software systems–a large-scale investigation," in *Proceedings of the 1st International Conference on Open Source Systems*, 2005, pp. 148–153.

[2] A. Deshpande and D. Riehle, "The total growth of open source," in *IFIP International Conference on Open Source Systems*, 2008, pp. 197–209.

[3] *2018 open source security and risk analysis (OSSRA)*, Synopsys, 2018. [Online]. Available: https://www.blackducksoftware.com/about/news-events/releases/audits-show-open-source-risks

[4] *The GitHub Blog - Thank you for 100 million repositories*, GitHub, 2018. [Online]. Available: https://github.blog/2018-11-08-100m-repos/

[5] H. Li, H. Kwon, J. Kwon, and H. Lee, "Clorifi: software vulnerability discovery using code clone verification," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 6, pp. 1900–1917, 2016.

[6] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 595–614.

[7] J. Jang, A. Agrawal, and D. Brumley, "ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (SP)*, 2012, pp. 48–62.

[8] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou *et al.*, "MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures," in *Proceedings of the 29th USENIX Security Symposium*, 2020, pp. 1165–1182.

[9] Y. Dong, W. Guo, Y. Chen, X. Xing, Y. Zhang, and G. Wang, "Towards the detection of inconsistencies in public security vulnerability reports," in *Proceedings of the 28th USENIX Security Symposium*, 2019, pp. 869–885.

[10] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *Proceedings of the 29th IEEE Symposium on Security and Privacy (SP)*, 2008, pp. 143–157.

[11] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016, pp. 1–16.

[12] K. Lu, M.-T. Walter, D. Pfaff, S. Nümberger, W. Lee, and M. Backes, "Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying." in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[13] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities," in *Proceedings of the 27th USENIX Security Symposium*, 2018, pp. 781–797.

[14] W. Wu, Y. Chen, X. Xing, and W. Zou, "KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities," in *Proceedings of the 28th USENIX Security Symposium*, 2019, pp. 1187–1204.

[15] I. Yun, D. Kapil, and T. Kim, "Automatic techniques to systematically discover new heap exploitation primitives," in *Proceedings of the 29th USENIX Security Symposium*, 2020.

[16] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[17] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[18] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang, "Understanding the reproducibility of crowd-reported security vulnerabilities," in *Proceedings of the 27th USENIX Security Symposium*, 2018, pp. 919–936.

[19] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software." in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2005, pp. 3–4.

[20] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, 2007, pp. 196–206.

[21] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proceedings of the 31st IEEE Symposium on Security and Privacy (SP)*, 2010, pp. 317–331.

[22] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[23] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[24] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[25] Pallor, *Ghostscript Null Pointer Dereference Vulnerability*, 2017 (accessed October 28, 2020). [Online]. Available: https://ghostscript.com/pipermail/gs-bugs/2017-January/046853.html

[26] *Backtraces*, 2020 (accessed December 2, 2020). [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/Backtraces.html

[27] Wikipedia, *File descriptor*, 2020 (accessed November 20, 2020. [Online]. Available: https://en.wikipedia.org/wiki/File_descriptor

[28] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, p. 190–200.

[29] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (SP)*, 2016.

[30] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "VulPecker: an automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC)*, 2016, pp. 201–213.

[31] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.