CAB-Fuzz: Practical Concolic Testing Techniques for COTS Operating Systems

Su Yong Kim, *The Affiliated Institute of ETRI;* Sangho Lee, Insu Yun, and Wen Xu, *Georgia Tech;* Byoungyoung Lee, *Purdue University;* Youngtae Yun, *The Affiliated Institute of ETRI;* Taesoo Kim, *Georgia Tech*

https://www.usenix.org/conference/atc17/technical-sessions/presentation/kim

This paper is included in the Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17).

July 12–14, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-38-6

Open access to the Proceedings of the 2017 USENIX Annual Technical Conference is sponsored by USENIX.

CAB-FUZZ: Practical Concolic Testing Techniques for COTS Operating Systems

Su Yong Kim*§ Sangho Lee† Insu Yun† Wen Xu†
Byoungyoung Lee¶ Youngtae Yun* Taesoo Kim†

*The Affiliated Institute of ETRI †Georgia Institute of Technology ¶Purdue University

Abstract

Discovering the security vulnerabilities of commercial off-the-shelf (COTS) operating systems (OSes) is challenging because they not only are huge and complex, but also lack detailed debug information. Concolic testing, which generates all feasible inputs of a program by using symbolic execution and tests the program with the generated inputs, is one of the most promising approaches to solve this problem. Unfortunately, the state-of-the-art concolic testing tools do not scale well for testing COTS OSes because of state explosion. Indeed, they often fail to find a single bug (or crash) in COTS OSes despite their long execution time.

In this paper, we propose CAB-FUZZ (Context-Aware and Boundary-focused), a *practical* concolic testing tool to quickly explore interesting paths that are highly likely triggering real bugs without debug information. First, CAB-FUZZ prioritizes the boundary states of arrays and loops, inspired by the fact that many vulnerabilities originate from a lack of proper boundary checks. Second, CAB-FUZZ exploits real programs interacting with COTS OSes to construct proper contexts to explore deep and complex kernel states without debug information. We applied CAB-FUZZ to Windows 7 and Windows Server 2008 and found *21 undisclosed unique crashes*, including two local privilege escalation vulnerabilities (CVE-2015-6098 and CVE-2016-0040) and one information disclosure vulnerability in a cryptography driver (CVE-2016-7219). CAB-FUZZ found vulnerabilities that are non-trivial to discover; five vulnerabilities have existed for 14 years, and we could trigger them even in the initial version of Windows XP (August 2001).

1 Introduction

Concolic testing is a well-known approach to *automatically* detect software vulnerabilities [8]. Empowered by its symbolic interpretation of the input, it generates and explores all feasible states in a program and thoroughly checks whether a certain security property can be violated. In particular, it has shown its effectiveness for *small* applications and/or applications with *source code*. For example, Avgerinos et al. [1] found more than 10,000 bugs in about 4,000 small applications. Also, Ramos and Engler [40] found 67 bugs in various open-source projects, such as BIND, OpenSSL, and the Linux kernel.

However, concolic testing *does not scale well for complex and large software* [5, 8, 13, 48], such as commercial off-the-shelf (COTS) operating systems (OSes). The complete concolic execution of COTS OSes would never terminate in a reasonable amount of time due to the well-known limitation of the symbolic execution, *state (or path) explosion*, where the number of feasible program states increases exponentially (e.g., once reaching a loop statement). Since the COTS OSes have massive implementation complexity, testing using symbolic execution ends up exploring a very small portion of program states, i.e., it cannot test deep execution paths.

Moreover, a proprietary nature of COTS OSes prevents concolic testing from exploring program states with *pre-contexts*. Unlike the open-source kernel for which the internal documentation and all test suites are publicly available [41, 50], COTS OSes do not provide such comprehensive information. Although manual annotation on the interface can help increase code coverage and detect logical bugs [27], it also does not scale. For these reasons, concolic execution on COTS OSes cannot explore program states that are only reachable after undergoing complex runtime operations.

In this paper, we propose CAB-FUZZ (Context-Aware and Boundary-focused), a practical system specialized to detect vulnerabilities in COTS OSes based on concolic testing. First, to overcome the scalability limitation of concolic testing, CAB-FUZZ *prioritizes* states likely having vulnerabilities. This prioritization is based on the observation that a majority of critical security bugs (e.g., memory corruption and information disclosure) originate

§This work is done while this author was a visiting scholar in Georgia Institute of Technology.

from *a lack of proper boundary checks*. This is why compilers and even hardware have adopted boundary-check mechanisms, such as SoftBound [37], SafeStack [28], and Intel Memory Protection Extensions (MPX) [20]. Therefore, we instruct CAB-FUZZ to generate and explore the boundary states of arrays and loops first, thereby detecting vulnerabilities as early as possible before exploding in terms of program states.

Second, to construct pre-contexts of COTS OSes without detailed debug information, CAB-FUZZ refers to *real programs* as a concolic-execution template. Since such a program frequently interacts with the COTS OSes to perform a certain operation, it embodies sufficient information and logic that constructs pre-contexts for using OS functions. Thus, if CAB-FUZZ runs a real program until it calls any target OS function that we are interested in, CAB-FUZZ is able to prepare with proper pre-contexts to initiate concolic testing correctly.

We implemented CAB-FUZZ based on a popular concolic testing tool, S2E [10], and evaluated it with two popular COTS OSes, Windows 7 and Windows Server 2008, especially for the 274 device drivers shipped with them. Since our approaches are general and independent of the OS, we believe they can be applied to currently unsupported OSes in the future.

In total CAB-FUZZ discovered *21 unique crashes* of six device drivers developed by Microsoft and ESET (§5). Among them we reported six reproducible crashes to Microsoft and one reproducible crash to ESET. Microsoft confirmed that three of them were undisclosed vulnerabilities and could be abused by a guest account for *local privilege escalation* (CVE-2015-6098 and CVE-2016-0040) and *information disclosure in a cryptography driver* (CVE-2016-7219). Especially, the later vulnerability even existed in the latest versions of Windows (Windows 10 and Windows Server 2016). Microsoft acknowledged the other three reports demanding administrator privilege and ESET fixed the bug we reported.

This evaluation result arguably demonstrates the effectiveness of CAB-FUZZ in finding vulnerabilities in COTS OSes despite its lack of completeness. CAB-FUZZ may not be able to trigger sophisticated bugs unrelated to boundary states. However, because of the fundamental scalability limitation of concolic testing, complete concolic testing is infeasible especially for large software. One of the contributions of CAB-FUZZ is that it changes the way we think of concolic testing—sacrificing completeness in a degree—to make it practical. Microsoft invests huge engineering efforts and computational resources in finding vulnerabilities, but CAB-FUZZ still discovered many different vulnerabilities in the Windows kernel using relatively moderate engineering efforts and computational resources. Specifically, we want to emphasize that Microsoft made fuzzing mandatory for every untrusted interface for every product, and their fuzzing solution has been running 24/7 since 2007 for a total of over 500 machine years [3]. However, despite this effort, CAB-FUZZ was able to discover 14-year-old bugs in Windows' kernel device drivers (§5.3).

This paper makes the following contributions.

- **Practical Techniques.** CAB-FUZZ makes concolic testing *practical* by addressing its two important challenges: state explosion and missing execution contexts. CAB-FUZZ prioritizes boundary conditions to trigger a crash before explosion and refers to a real application to construct proper execution contexts.
- **Evaluation and In-depth Analysis.** We analyzed the implementation of COTS OSes in detail to figure out why CAB-FUZZ was able to detect their vulnerabilities effectively compared to conventional techniques.
- **Real-world Impact.** CAB-FUZZ discovered *21 unique crashes* of device drivers for Windows 7 and Windows Server 2008. We reported all reproducible crashes to the vendors. They confirmed that four of the reported crashes were critical and fixed them. Specifically, two of them were privilege escalation vulnerabilities and one was an information disclosure vulnerability in a cryptography driver.

The rest of this paper is organized as follows. §2 describes the challenges of performing concolic testing for COTS OSes. §3 depicts CAB-FUZZ and §4 describes its implementation. §5 evaluates CAB-FUZZ's vulnerability-finding effectiveness. §6 discusses the various aspects of CAB-FUZZ including its limitations, and §7 presents related work. §8 concludes the paper.

2 Challenges for COTS OSes

This section elaborates on the challenges involved in performing concolic testing for COTS OSes to clearly motivate our proposed system, CAB-FUZZ.

2.1 Binary

Automated binary analysis is necessary for production software (e.g., COTS OS) because (1) it usually contains third-party binaries and libraries without source code, (2) its behavior can be changed due to compiler optimization or linking, and (3) its code can be written with multiple programming languages, making source code analysis difficult. However, the following two challenges make concolic testing for COTS OSes unpractical.

Missing Documentation and Test-suites. When doing automated testing, especially for COTS OSes, a lack of source code and document is a critical hurdle because most of the communication interfaces between user- and kernel-space are undocumented (often intentionally) and vary dramatically across versions [21]. Further, COTS

```
1  // global arrays
2  bool flag_table[125];
3  void (*fn_table[36])(); //function pointer array
4
5  int dispatch_device_io_control(unsigned long ctrl_code,
6                                 unsigned long *buf) {
7    switch (ctrl_code) {
8    case 0x8fff23cc:
9    case 0x8fff23c8:
10     // sanitizing conditions (simplified)
11     if (buf[0] > 246 || buf[1] > 124 || buf[2] > 36)
12         return -1;
13
14     if (flag_table[buf[1]]) {
15       // buf[2] == 36 -> out-of-bound access
16       (*fn_table[buf[2]])();
17     }
18
19     for (int i = 1; i <= buf[0]; ++i) { ... }
20
21   // NOTE. the below included to comprehensively illustrate
22   // the effectiveness of on-the-fly technique.
23   // Not exist in the original NDProxy
24   case 0x8fff23c4:
25     // set all elements of flag_table to true
26     for (int i = 0; i < 125; i++)
27       flag_table[i] = true;
28     ...
29   }
30 }
```

Figure 1: A simplified code snippet reconstructed from NDProxy vulnerability (CVE-2013-5065) [11]. It resulted in a local privilege escalation in Windows XP and Server 2003.

OSes often do not provide test suites such that it is difficult to generate proper input values that pass input validation routines at an early state. This prevents the concolic testing procedure from reaching later and deeper stages. Even the state-of-the-art techniques (S2E [10] and Dowser [19]) rely on unit tests to pass input validation routines.

Handling Symbolic Memory. There are two common ways to handle symbolic memory in concolic testing: treating it as a symbolic array (*symbolization*) or concretizing it (*concretization*). Memory symbolization is typically used to avoid the state explosion problem because it efficiently abstracts the execution state. However, memory symbolization is not suitable for a COTS binary because it heavily uses the static information (e.g., object size) for performance optimization, which is often unavailable. Further, it produces complex constraints that are barely solvable in large-scale, real-world software.

Therefore, CAB-FUZZ concretizes every symbolic memory as it produces solvable constraints even for large-scale software. But, it has to cope with the state explosion problem as we discuss in the next section.

2.2 State Explosion

We illustrate state explosion with an NDProxy vulnerability (CVE-2013-5065) and S2E [10].

CVE-2013-5065. Figure 1 shows a simplified code snippet reverse-engineered from the NDProxy kernel driver. The `dispatch_device_io_control` function handles the requests of a user-mode process. `ctrl_code` and `buf` are inputs from a user-mode process, where `ctrl_code` represents an operation and `buf` contains user data.

According to our analysis, this vulnerability originated from the misverification of `buf[2]` at Line 11. `buf[2]` is used as an index to refer to `fn_table` and it should lie between `0` and `35` to avoid memory access violations. In principle, having `ctrl_code` and `buf` as symbolic variables, S2E [10] is supposed to identify the offending input satisfying the vulnerable condition. However, we found it suffers from state explosion.

State Explosion Problem. We carefully adjusted S2E to check the code (Figure 1) as a preliminary experiment (§5.1). Due to state explosion, it took *two hours* while consuming up to 15 GB of memory to detect the vulnerability. First, S2E explored all feasible paths of *symbolic memory*—a memory region a symbolic variable controls. The code had at least two symbolic memory arrays: `fn_table` and `flag_table`, where `fn_table` generated 37 states due to the condition of `buf[2]` at Line 11, and `flag_table` generated 125 states due to the condition of `buf[1]` at Line 11. Second, S2E explored all possible paths of a *loop* controlled by a symbolic variable. This code had a loop controlled by `buf[0]` at Line 19, generating at least 247 states in our observation. In total, S2E generated more than a million states just for two symbolic memories and a single loop.

Exploring all feasible paths of a program is difficult in practice due to state explosion. Instead, CAB-FUZZ prioritizes interesting paths that more likely trigger vulnerabilities. For example, the vulnerability in Figure 1 is triggered when `buf[2]` has the upper-bound value `36`. Focusing on such *boundary states* allows us to detect many vulnerabilities while avoiding state explosion (§3.2).

2.3 Missing Execution Contexts

To avoid state explosion, concolic testing tools need to check individual functions instead of the entire program from the beginning. However, functions can have close relationships with each other such that we cannot establish proper contexts when skipping some of them (e.g., a function for initializing shared variables) [40].

Figure 1 also shows a crash example that context-unaware concolic testing tools cannot detect (Lines 14, 16, and 27). In fact, `fn_table[buf[2]]` will be executed only after `dispatch_device_io_control` with `0x8fff23c4` as `ctrl_code` has been called first since it depends on a global array `flag_table`. When testing such a function, existing concolic testing tools just treat its input parameters as symbolic variables, ignoring context such as the sequence of function calls. However, this cannot generate a crash because no elements of `flag_table` have the value required for the crash. Therefore, existing tools cannot detect the bug in our example.

CAB-FUZZ targets COTS OSes such that it aims to solve this problem without relying on any prior knowl-

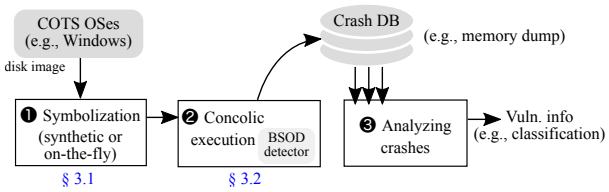

Figure 2: An overview of CAB-FUZZ's workflow.

edge (e.g., annotation). Our basic idea is to run a *real program*, instead of a synthetic program, to let it construct pre-contexts. Later, when the program is calling a target function, CAB-FUZZ initiates concolic testing *on-the-fly*. This allows us to get enough pre-contexts to test the target function with minimal overhead (details are in §3.1.2.)

3 Design

In this section, we describe in detail CAB-FUZZ's design and the techniques that allow for concolic execution for COTS OSes. CAB-FUZZ is a full-fledged vulnerability-detection system for COTS binaries, and in particular, it aims to make concolic testing (see §2) practical in the context of COTS OSes.

Figure 2 depicts an overview of CAB-FUZZ. First, it takes a disk image of the targeted COTS OS as an input. Then, it determines when to start symbolic execution either by synthetic symbolization (§3.1.1) or on-the-fly symbolization (§3.1.2). After deciding what to symbolize, CAB-FUZZ performs the concolic testing. In order to address the state explosion problem, CAB-FUZZ employs two new techniques, namely, array-boundary prioritization (§3.2.1) and loop-boundary prioritization (§3.2.2), which focus on boundary states (§3.2). Once CAB-FUZZ observes a kernel crash during the symbolic execution, it attempts to generate concrete input and a crash report to help reproduce the observed crash.

3.1 Symbolization for Kernel

The goal of CAB-FUZZ is to detect the vulnerabilities in the kernel using concolic testing. In particular, CAB-FUZZ symbolizes a certain memory location during kernel execution such that any instruction involving this location is symbolically executed. Although this procedure resembles generic concolic testing methods, we specialize CAB-FUZZ for handling COTS OSes by considering two important issues: *when to start the symbolic execution* and *what memory regions to symbolize*. The kernel can be considered as a long-running process or system service, and the majority of its functional components depend on previous kernel execution states. CAB-FUZZ takes two different approaches in this regard: synthetic symbolization (§3.1.1) and on-the-fly symbolization (§3.1.2). Overall, synthetic symbolization launches a previously built user-space program and explicitly starts the symbolic execution phase. On the other hand, on-the-fly symbolization retrofits the existing user-space programs to better construct the legitimate kernel execution contexts and seamlessly starts the symbolic execution at a certain execution point.

3.1.1 Synthetic Symbolization

Synthetic symbolization launches a previously built user-space program that initiates the symbolic execution. This largely follows previous concolic execution techniques in that CAB-FUZZ also launches synthetic programs to start the symbolic execution phase. The key difference is that CAB-FUZZ tailors the user-space programs to test kernel device drivers. Our synthetic program invokes a function controlling an IO device (i.e., `NtDeviceIoControlFile`) while symbolizing its parameters.

Figure 3 shows example code to test `NtDeviceIoControlFile`. In particular, for each device driver, we obtain the corresponding device driver handle at Line 17. Using this handle, CAB-FUZZ invokes `NtDeviceIoControlFile` while symbolizing the two parameters, `ctrl_code` and `in_buf`, which primarily control the behavior of a device driver (see Figure 1). We observed that symbolizing the size of `in_buf` resulted in state explosion, leading us to decide not to symbolize it (explained later). The memory symbolization is carried out by utilizing existing runtime helper functions in the concolic execution engine (i.e., `s2e_make_symbolic`). Once these two parameters are symbolized, CAB-FUZZ symbolically interprets these parameters while executing `NtDeviceIoControlFile`.

State Explosion due to Input Buffer Size Symbolization. We explain why input buffer size symbolization generates state explosion. Windows provides three methods to deliver a user-space input buffer to the kernel, configured using the lowest two bits of `ctrl_code` [33]. The first method, *buffered I/O*, allocates a kernel memory buffer whose size is the same as that of a user input buffer and copies the input buffer's content to the kernel buffer. The buffered I/O, however, generates state explosion, as shown in Figure 4. At Line 9, `in_buf_size` is used as a condition of the for loop, so it generates `0x7FFF0000` states even with the constraint at Line 7.

The other two methods (*direct I/O* and *neither buffered nor direct I/O*) do not directly generate state explosion since they let a kernel device driver access the user buffer via a memory descriptor list (MDL) or virtual address. However, since we focus on COTS OSes, we do not know which method a target driver uses to access a user input buffer. Consequently, CAB-FUZZ should symbolize the for loop no matter which method the target driver uses.

3.1.2 On-the-Fly Symbolization

As shown in §2.3, existing concolic testing tools cannot check individual target functions due to the lack of context awareness. To this end, on-the-fly symbolization retrofits the real user-space programs to better construct

```
HANDLE device_handle;
unsigned long in_buf[BUF_SIZE] = {0};
unsigned long out_buf[BUF_SIZE] = {0};
unsigned long ctrl_code = 0;
NTSTATUS status;
UNICODE_STRING device_name;
OBJECT_ATTRIBUTES object_attributes;
ACCESS_MASK max_allowed_access;
IO_STATUS_BLOCK io_status_block;

// get maximum access allowed for the target device driver
max_allowed_access = get_allowed_access(&device_name);

obect_attributes.ObjectName = &device_name;

// get handle of the target device driver
status = NtCreateFile(&device_handle, max_allowed_access,
                      &object_attributes, ...);

if (status)
  return -1; //cannot get a handle

// initate concolic execution and symoblize params
cab_start_concolic_testing();

s2e_make_symbolic(&ctrl_code, sizeof(ctrl_code), "code");
s2e_make_symbolic(&in_buf, sizeof(in_buf), "buf");

// targeted call
NtDeviceIoControlFile(
  device_handle,    // handle to target device
  NULL,             // A handle to an event
  NULL,             // ApcRoutine procedure
  NULL,             // a pointer to pass to ApcRoutine
  &io_status_block, // receive the final completion status
  ctrl_code,        // a control function to be executed
  &in_buf,          // input buffer
  BUF_SIZE,         // input buffer size
  &out_buf,         // output buffer
  BUF_SIZE);        // output buffer size

// terminate and generate a testcase
s2e_kill_state(0, "Successfully done");
```

Figure 3: Example code of the synthetic symbolization testing the `NtDeviceIoControlFile` function (see §3.1.1 for explanation). `get_allowed_access()` is related to the access permission per the driver (see §4.1 for more details). The prototype of `NtDeviceIoControlFile` function can be found in [35].

the legitimate kernel execution contexts and seamlessly starts the symbolic execution at a certain execution point. Specifically, unlike existing concolic testing tools, our on-the-fly concolic testing tries to satisfy the *pre-contexts* of a function to crash in our best effort by following the real execution procedure of a COTS binary, as shown in Figure 5. It (1) runs and monitors the execution of a user-space program, (2) lets the program and kernel construct pre-contexts, (3) monitors input values to a target function and selects some of them, and (4) performs runtime concolic testing while designating the selected values as symbolic variables.

For example, to test `dispatch_device_io_control` in Figure 1, our tool runs a user-space program containing the code, ensuring the initialization has been called (i.e., `ctrl_code = 0x8fff23c4`) before other `NtDeviceIoControlFile` calls. Since the pre-context is now fulfilled, concolic testing can automatically generate the value that results in a crash.

```
// ctrl_code, in_buf_size, and in_buf are given from
// a user-space process. kernel_mem is a kernel-space buffer

#define USER_ADDR_MAX 0x7fff0000

if (ctrl_code & 3 == 0) { // Buffered I/O
    if (in_buf_size < USER_ADDR_MAX) {
        ...
        for (int i = 0; i < in_buf_size; i++) {
            kernel_mem[i] = in_buf[i];
        }
        ...
    }
}
```

Figure 4: Pseudo code showing why symbolizing an input buffer size generates state explosion during concolic testing.

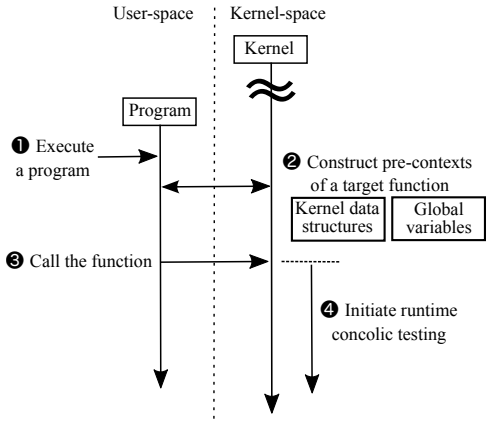

Figure 5: Overall procedures of on-the-fly concolic testing: ❶ CAB-FUZZ executes a real user-space program; ❷ CAB-FUZZ lets the program and kernel interact with each other to construct the pre-contexts of a target function; ❸ the program calls a target function; ❹ CAB-FUZZ hooks the event and initiates runtime concolic testing from this point.

Furthermore, our on-the-fly concolic testing method can work with COTS binaries that provide only partial information. Many COTS binaries lack full documentation, so we cannot obtain all the information to test target functions. This makes existing concolic testing tools ineffective in practice because it is difficult to pass the sanitization routines without satisfying basic conditions among inputs. Even in such a case, our on-the-fly symbolization has a chance to bypass uninteresting sanitization routines, yet effectively test the target function by deriving input conditions from a real execution [46].

3.2 Boundary-state Prioritization

In this section, we introduce *boundary-state prioritization* that attempts to overcome the state explosion due to symbolic arrays and loops in COTS OSes. The key idea of the boundary-state prioritization is to defer the analysis of uninteresting states based on the likelihood of security vulnerability (e.g., memory corruption and disclosure). In other words, we focus on triggering security vulnerabilities via concolic execution while compromising the completeness of testing for performance and scalability.

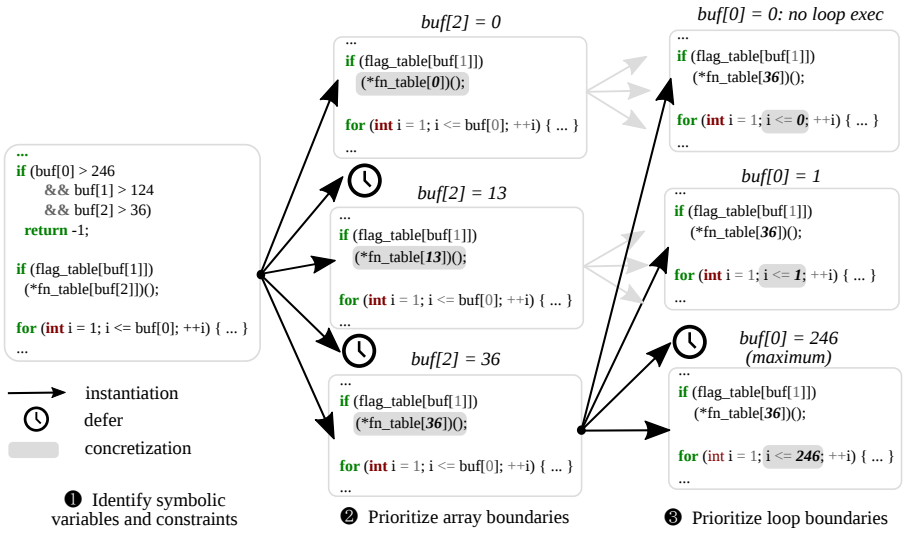

Figure 6: Overall procedures of boundary-state prioritization: ❶ CAB-FUZZ identifies symbolic variables and constraints; ❷ CAB-FUZZ selectively concretizes symbolic memories according to their boundary information; ❸ CAB-FUZZ selectively concretizes loops using their boundary information (in this example, we do not symbolized `flag_table` for simplicity.)

Figure 6 shows the overall procedures. First, it figures out constraints that limit the range of symbolic variables using KLEE's range analysis function [6]. Second, it detects symbolic memories controlled by the symbolic variables and selectively concretizes them according to their boundary information (*array-boundary prioritization*). Third, it detects loops and selectively iterates through them using the boundary information (*loop-boundary prioritization*). Without our prioritization techniques, the total number of states exponentially increases according to the number of symbolic memories and loops. If the number of symbolic arrays and loops is n and the number of possible states of each symbolic array or loop is s_i, the total number of states to explore will be $\prod_{i=1}^{n} s_i$. In contrast, our techniques test $\prod_{i=1}^{n} c = c^n$ states first, where c is a constant.

3.2.1 Array-boundary Prioritization

We explain our array-boundary prioritization technique with two symbolic memories `flag_table` and `fn_table` in Figure 1 and Figure 6. As we discussed in §2.2, `flag_table` generates 125 states and `fn_table` generates 37 states, which result in $125 \times 37 = 4,625$ states total.

Exploring all states is challenging, especially when the length of a target array is long and/or many symbolic memories and loops are associated with it. Instead, CAB-FUZZ drives symbolic execution to visit boundary cases first, which highly likely have problems. Specifically, CAB-FUZZ creates two states for each symbolic memory by solving the associated constraints: the *lowest* memory address and the *highest* memory address. Note that the two boundary states could result in exceptions due to crashes or boundary checks. To proceed the test, CAB-FUZZ additionally creates a state for an arbitrary memory address between them.

The second step of Figure 6 shows array-boundary prioritization for `fn_table`. CAB-FUZZ prioritizes three states according to the associated symbolic variable's constraints: the lowest memory address `fn_table[0]`, the highest memory address `fn_table[36]`, and an arbitrary memory address between them, e.g., `fn_table[13]`.

3.2.2 Loop-boundary Prioritization

Handling a loop can result in state explosion [18]. To avoid it, CAB-FUZZ limits the number of state forks at the same loop to focus on boundary states. Specifically, it focuses on only three states: a state with *no loop execution*, a state with *a single loop execution*, and a state with *the largest number of loop executions*. Figure 6 has a loop whose number of iterations depends on `buf[0]`. Since its values lie between 0 and 246, this loop generates 247 states. To avoid such state explosion, our loop-boundary prioritization method focuses on three kinds of loop executions: 0, 1, and maximum (246) times.

In total, our method generates only 27 states first, `flag_table` (3) × `fn_table` (3) × the loop (3), including the boundary condition causing a crash, `buf[2] == 36`.

4 Implementation

We implemented CAB-FUZZ by extending S2E [10]. In particular, we focused on crashing *Windows device drivers*, which are popular and complex commodity COTS kernel binaries. In total, we wrote around 2,000 lines of new code (mixed with C/C++, Lua, and Python).

4.1 Synthetic Symbolization

We used `NtCreateFile` to obtain the handlers for device drivers. As opposed to using the typical `CreateFile`,

this approach allowed us to access all device drivers, including those of all internal and undocumented devices.

When opening or creating a file object using `NtCreateFile`, we can specify 13 different access rights for the file object [34]. Since we aimed to obtain and test as diverse access rights as possible, we repeatedly invoked `NtCreateFile` in `get_allowed_access` to obtain all possibly allowed permission accesses.

4.2 On-the-fly Symbolization

Target API. To detect device driver bugs with on-the-fly symbolization, we interpose the `NtDeviceIoControlFile` function, which is the lowest user-level internal API for communicating with the kernel devices. Any user-space process attempting to access a device driver eventually calls the function, so hooking it allows us to test all the device drivers used during the on-the-fly symbolization phase. The below half of Figure 3 shows the specification of `NtDeviceIoControlFile`, and CAB-FUZZ symbolizes `in_buf` and `ctrl_code` on-the-fly.

Fulfilling Pre-context. We inferred the pre-context of `NtDeviceIoControlFile` by running real user-space programs using this function during their normal execution. We tried to find such programs with an assumption: system management and antivirus software would use it because they frequently access device drivers. Finally, we found 16 programs (e.g., `dxdiag.exe` and `perfmon.msc`) accessing 15 different drivers (e.g., KsecDD and WMIDataDevice) during their execution[1]. We used these target programs to test the corresponding device drivers during the on-the-fly symbolization phase (§5.2).

4.3 Boundary-state Prioritization

Prioritizing Array Boundaries. For a symbolic memory array, CAB-FUZZ estimates its lower and upper boundary addresses and one arbitrary address between them. CAB-FUZZ uses the `getRange` method of the `klee::Solver` to compute these boundary addresses [6]. This method receives an expression as input and returns a pair of the minimum and maximum values of the expression. Since `getRange` is computationally heavy, instead of invoking this function in every symbolic memory access, CAB-FUZZ proceeds only if the targeted memory has triggered a state forking at one point in the past. Specifically, if state forking has never been triggered, CAB-FUZZ does not perform any prioritization for the memory, as we found that such memory usually has only one concrete value. If the state forking is triggered at the same location, CAB-FUZZ performs prioritization when it observes the memory again later.

Prioritizing Loop Boundaries. CAB-FUZZ focuses on three states of each loop: no, single, and maximum execution (§3.2.2). However, identifying how many times a loop will be executed is difficult because it varies according to input variables and compiler optimization techniques (e.g., loop unrolling [45]). We develop a practical loop-boundary prioritization technique that does not suffer from variable loop conditions. Whenever CAB-FUZZ encounters a loop, it first generates two forking states: no and single iteration of the loop. Then, to get the maximum number of loop executions, it repeatedly forks and kills states until it observes the last state forking, which would be the maximum because CAB-FUZZ concretely and sequentially executes the loop until it terminates. During state forking, CAB-FUZZ does not call the solver to minimize overhead; it calls the solver only when generating test cases. Also, we confirmed that killing unnecessary loop states had negligible performance overhead.

[1] Due to the space limit, we do not enumerate all of them.

4.4 Analyzing Crashes

CAB-FUZZ generated many inputs that crashed the Windows kernel, but a large portion of them may not be unique vulnerabilities that require in-depth analysis. A typical technique of classifying such crashes is to inspect the call stack at the time of the crash, but it is difficult to identify stack information without debug symbols. More seriously, we found that many memory access violations are delegated to the default exception handler, making it even harder to uniquely identify the call stack information of the kernel thread that actually raised the exception.

To solve this problem, CAB-FUZZ records and inspects the blue screen of death (BSOD) information when the Windows kernel executes the `KeBugCheck*` function to gradually bring down the computer [32]. Specifically, CAB-FUZZ uses the function's *BugCheckCode* value representing a BSOD reason and *instruction address* where the exception occurred to differentiate crashes. CAB-FUZZ treats two crashes as different when (1) they have different BugCheckCode values or (2) they have the same BugCheckCode value, but their instruction addresses belong to different functions.

5 Evaluation

We evaluate the effectiveness of CAB-FUZZ in finding security vulnerabilities in the Windows device drivers. Table 1 summarizes all new unique crashes discovered by CAB-FUZZ. In general, our evaluation consists of two categories targeting synthetic symbolization (§5.1) and on-the-fly symbolization (§5.2). In particular, our evaluation aims at answering the following questions:

- Per synthetic symbolization, how efficiently did CAB-FUZZ detect the known vulnerability (Figure 1) compared to the conventional concolic testing tool? (§5.1.1)
- Per synthetic and on-the-fly symbolization, how many new unique crashes did CAB-FUZZ discover? (§5.1.2 and §5.2.1)

	# of Crashes			
	Total	**Synthetic (Prioritization)**		**On-the-fly**
		Off	**On**	
NDIS[†,§]	11	5	10	-
SrvAdmin[†,§]	4	4	4	-
NSI[§]	2	2	2	0
ASYNCMAC[†,§]	1	1	1	-
FileInfo[§]	2	0	0	2
ehdrv[†,§]	1	0	0	1
Total	**21**	**12**	**17**	**3**

[†]: Windows 7, [§]: Windows Server 2008

Table 1: The list of newly discovered unique crashes by CAB-FUZZ among the 274 drivers we tested. The total number of discovered unique crashes is smaller than the summation of the other three columns (two synthetic and one on-the-fly cases) because we removed duplicate crashes and only counted the unique crashes.

- Per synthetic and on-the-fly symbolization, what particular characteristics did newly discovered crashes exhibit? (§5.1.3 and §5.2.2)

Experimental Setup. Our experiments were performed on 3 GHz 8-core Intel Xeon E5 CPU with 48 GB of memory. We ran CAB-FUZZ with the latest versions of Windows 7 and Windows Server 2008 as of April 2016. For example, two of the drivers for which CAB-FUZZ found crashes, NDIS and SrvAdmin, were updated in December 2015 and October 2015, respectively. The detailed configuration setting for CAB-FUZZ is further described in each subsection if required.

5.1 Synthetic Symbolization

To show the effectiveness of the synthetic symbolization and boundary prioritization techniques, we carried out the following two experiments. First, to see if the implementation of CAB-FUZZ can address the challenges (especially in handling state explosion), we applied boundary-state prioritization techniques to the known NDProxy vulnerability and compared the result before applying (§5.1.1). Next, we describe our experiences in applying CAB-FUZZ to discover new crashes in the Windows kernel driver using synthetic symbolization techniques (§5.1.2). Further, we manually analyzed all unique crashes newly discovered by CAB-FUZZ (§5.1.3).

Configuration. We configured CAB-FUZZ to target 186 and 88 kernel device drivers on Windows 7 and Windows Server 2008, respectively (274 drivers in total). Among them, CAB-FUZZ detected *six device drivers* with 21 unique crashes (Table 1). For each device driver, we specified `ctrl_code` and `in_buf` as symbolic variables (shown in Figure 3). It is worth nothing that due to the space limit of this paper, we have only presented the results with a random search strategy, which showed the best performance overall compared to other depth-first and breadth-first search strategies. Since the random search algorithm may produce different evaluation results due to its random nature, we ran it five times per evaluation and computed the average. In addition, when we found the same crash of the same driver in Windows 7 and Windows Server 2008, we further tested it in Windows 7 only since it is the recent version.

Prioritization	**Time (s)**	**#States**
None	7,196	384,817
Loop boundary	516	30,604
Array boundary	2	78
Both	2	78

Table 2: The effectiveness of boundary-state prioritization techniques (based on the synthetic symbolization) to detect the NDProxy vulnerability: **Time** shows the elapsed time and **#States** shows the number of explored states to detect the vulnerability.

5.1.1 Detecting Known Vulnerability

We measured the time taken to find the NDProxy vulnerability (Figure 1) before and after applying the prioritization techniques. We also measured the number of program states that need to be explored to find the vulnerability.

When both array- and loop-boundary prioritization techniques were applied, CAB-FUZZ found the NDProxy vulnerability within 2 seconds (Table 2). It took 2 seconds with the array-boundary prioritization and 516 seconds with the loop-boundary prioritization if each technique was individually applied. The array-boundary prioritization is more effective than the loop-boundary prioritization in the case of the NDProxy vulnerability because the state related to the crash (i.e., `buf[2] == 36`) is quickly created by the array-boundary prioritization technique, as shown in Figure 1.

However, when none of prioritization techniques were applied, it took 7,196 seconds to find the vulnerability. This significant slowdown is caused by the huge number of states that need to be covered in order to find the vulnerability—384,817 states in total, which is 4,934 times larger than the number of states when both were applied.

5.1.2 Newly Discovered Crashes

To determine the effectiveness of our synthetic symbolization with and without prioritization techniques, we applied CAB-FUZZ to all kernel device drivers in Windows 7 and Windows Server 2008. In total, CAB-FUZZ found 18 new unique crashes from four different device drivers, as shown in Table 1. Specifically, the prioritization techniques allowed CAB-FUZZ to detect six more unique crashes while missing one unique crash. Thus, we believe this technique is effective in practice.

Driver		No prioritization			Prioritization		
	#Crash	Time (s)	#States	Mem. (MB)	Time (s)	#States	Mem. (MB)
NDIS	1	837	151	5,537	287	58	4,813
	2	871	156	5,545	467	86	4,971
	3	1,763	271	7,690	617	124	5,027
	4	5,066	637	14,946	824	171	5,461
	5	8,682	1,180	22,768	1,202	214	6,093
	6	-	-	-	1,930	306	7,980
	7	-	-	-	4,381	586	9,781
	8	-	-	-	4,977	637	10,376
	9	-	-	-	5,018	642	10,377
	10	-	-	-	6,056	704	10,893
SrvAdmin	1	1	23	4,321	2	23	4,325
	2	3	54	4,359	6	71	4,401
	3	51	126	4,464	51	126	4,476
	4	1,892	2,319	15,321	657	953	5,390
NSI	1	1	2	4,356	1	2	4,357
	2	1,951	7,622	5,979	1,092	1,952	5,843

Table 3: Detailed experiment results of the four kernel device drivers tested by CAB-FUZZ with and without prioritization techniques: **#Crash** represents how many crashed observed during experiments; **Time** represents the elapsed time; **#States** represents the number of explored states; and **Memory** represents the consumed memory to detect each crash. All values are averaged over five runs.

5.1.3 Effectiveness of Boundary-state Prioritization

To clearly understand the effectiveness of our prioritization techniques, we manually analyzed why CAB-FUZZ without our prioritization techniques cannot detect the six unique crashes and what is the root cause of its slowdown. Note that our prioritization techniques were ineffective to ASYNCMAC (elapsed time and memory consumption were almost the same,) so we skipped analyzing it in depth. Also, we were not able to test their effectiveness with other device drivers because CAB-FUZZ was not able to detect their crashes. Table 3 represents how many crashes were observed during our evaluation along with elapsed time, the number of tested states, and consumed memory. All results are averaged over five runs. Note that, because we use a random search strategy, it is difficult to directly compare each crash.

NDIS. The six crashes that the prioritization technique detected were due to input buffers whose values were used as offsets of a symbolic array. When there were no routines to check the range of input buffer values or the values were incorrect, crashes were generated due to invalid offsets. However, without prioritization, CAB-FUZZ was unable to reproduce it due to memory exhaustion.

Among the five crashes that CAB-FUZZ with prioritization was able to generate but CAB-FUZZ without prioritization was unable to do, we explain a crash at `ndisNsiGetNetworkInfo` function of ndis.sys in detail. The function had a symbolic memory array using `in_buf[5]` as an offset, but did not have any routine to check its value. As a result, when the symbolic array pointed to invalid memory and there was a write attempt to the memory, a crash occurred. This happened when the value of `in_buf[5]` was at the boundary condition: whether it was larger than or equal to `0xbc0`, but, without prioritization, CAB-FUZZ could not generate this state due to a lack of available memory (it concretized ∼30 values of `in_buf[5]` before termination.)

On the other hand, the single crash that CAB-FUZZ with prioritization could not detect was due to the loop-boundary prioritization technique. We found that the `ndisNsiGetInterfaceRodEnumObject` function of ndis.sys generated a crash when it ran a loop four times with a specific condition. Note that our loop-boundary prioritization technique runs a loop 0, 1, or a maximum number of times, so it cannot cover such a specific case. To confirm it, we applied CAB-FUZZ only with the array-boundary prioritization to NDIS. We could trigger the specific case also, though it took about one hour longer.

SrvAdmin. We analyzed SrvAdmin and confirmed that the 2.9× slowdown of CAB-FUZZ without prioritization was due to the state explosion caused by a specific loop located at the `SvcAliasEnumApiHandler` function of srvnet.sys. This loop was not related to the crash we found, but it generated 8,285 states that were approximately 20% of the entire states (41,279) of SrvAdmin. With the loop-boundary prioritization, CAB-FUZZ could postpone less important states, so it detected the crash earlier.

NSI. We analyzed NSI and confirmed that our prioritization techniques made CAB-FUZZ detect the two unique crashes 1.8× faster. While symbolic arrays or loops were not directly related to these crashes, we found that prioritization techniques helped concolic testing avoid the state explosion, so that it kept exploring the program states and finally reached the vulnerable program state.

5.2 On-the-Fly Symbolization

We evaluate the effectiveness of on-the-fly symbolization. We summarize the new crashes the on-the-fly technique detected (§5.2.1) and analyze them in detail to show how this technique was able to detect them (§5.2.2).

5.2.1 Newly Discovered Crashes

Overall, CAB-FUZZ identified three unique crashes using on-the-fly symbolization (Table 1). Note that the crashes found by the two techniques were not overlapped because (1) the on-the-fly technique was unable to test some drivers (NDIS, SrvAdmin, and ASYNCMAC) because we had no reference applications accessing them and (2) some crashes (in NSI) were triggered only if they had improper pre-contexts. Therefore, we believe both techniques are complementary to each other.

5.2.2 Effectiveness of On-the-fly Symbolization

To figure out how the on-the-fly technique helps find a vulnerability, we manually analyzed three crashes that CAB-FUZZ found in FileInfo and ehdrv device drivers.

	# of Crashes		
	Total	**Synthetic**	**On-the-fly**
WMIDataDevice	2	1	1
TCP	3	3	0
Total	**5**	**4**	**1**

Table 4: The crashes of Windows XP CAB-FUZZ found.

FileInfo. We found two reasons why the on-the-fly technique was able to find these cases and why synthetic symbolization was not. First, FileInfo was loaded only when a certain application started (e.g., `perfmon.msc`). Second, FileInfo sanitized an input buffer size at an early stage; it should be 12. Running `perfmon.msc` satisfied both conditions for the on-the-fly technique, but a synthetic program was unable to do that.

ehdrv. ehdrv was a third-party driver installed by ESET Smart Security 9, which was used by `SysInspector.exe` of the vendor. The on-the-fly technique detected a memory corruption crash of ehdrv on Windows 7 by running `SysInspector.exe` before symbolization. In contrast, the synthetic technique cannot detect it because ehdrv had a security feature: it was only accessible by an authorized process like `SysInspector.exe`, which cannot be satisfied by a synthetic program.

5.3 Fourteen-Year-Old Bugs

We applied CAB-FUZZ to the latest version of Windows XP (April 2014) and found five unique crashes (Table 4). Among them, a crash of WMIDataDevice and all three crashes of TCP were also observed in the initial version of Windows XP (August 2001), implying *nobody detected them for about 14 years*.

6 Discussion

In this section we explain some limitations of CAB-FUZZ.

Boundary-state Prioritization. Our boundary-state prioritization methods assume that the symbolic memory under consideration stores data such that values between boundaries are less important; that is, we sacrifice some completeness for efficient detection. However, if the symbolic memory is related to control flow (e.g., jump table and virtual function table), we should consider all the values to maintain code coverage. To solve this problem, we plan to adopt static analysis in our system. Whenever it detects a symbolic memory array, it performs static analysis to know whether the symbolic array stores instruction addresses for indirect calls or jumps. In such a case, it checks all the values of the symbolic array to enhance code coverage. Also, our methods cannot handle data structures with undefined size. We plan to enhance CAB-FUZZ to support this in the future. For example, we can adopt UC-KLEE [14, 39, 40]-like approaches.

On-the-fly Symbolization. Our on-the-fly approach is a best-effort approach. If we cannot find programs constructing pre-contexts for vulnerable functions, it cannot crash them. Thus, this approach is not suitable for detecting the security vulnerabilities of rarely used functions. To detect vulnerabilities in such functions, one would need to run synthetic and on-the-fly testing in parallel.

Manual efforts. Currently, we manually specify a target API, `NtDeviceIoControlFile`, for the both synthetic and on-the-fly symbolizations, and programs constructing pre-contexts for the on-the-fly symbolization. In the future, we will explore how to automate both phases for enhancing CAB-FUZZ's scalability.

7 Related Work

In this section, we introduce previous work related to CAB-FUZZ. Among a large number of studies on symbolic and concolic execution, we focus on four research topics closely related to CAB-FUZZ: (1) binary-level symbolic execution, (2) kernel and device driver testing, (3) boundary value analysis, (4) overflow detection, and (5) lazy initialization.

Binary-level Symbolic Execution. Symbolic execution was originally designed to work with source code [4, 7, 12, 16, 29, 30], and extended to test binary programs lacking source code and detailed debug information (e.g., proprietary software and malware). SAGE [3, 17] is the earliest effort to apply symbolic execution to binary programs and many schemes such as SmartFuzz [36], LESE [42], IntScope [47], S2E [10], FuzzBALL [2, 31], Mayhem [9], MegaPoint [1], and DIODE [44] follow it. Among them, only S2E and FuzzBALL are designed to test OS kernels, while FuzzBALL does not support Windows binaries. Consequently, S2E is the only scheme that we can directly compare with CAB-FUZZ.

Kernel and Device Driver Testing. CAB-FUZZ is designed to test COTS OSes and device drivers. To the best of our knowledge, only a few studies apply concolic execution to OSes and device drivers. Yang et al. [50] use their EXE system [7] to create a symbolic disk for Linux file system testing. Their system relies on file system code instrumentation to create the symbolic disk, so it cannot be applied to COTS OSes directly.

DDT [27] is a QEMU-based system to test closed-source binary device drivers for Windows, which became a part of S2E [10]. It can test device drivers without real hardware by creating symbolic hardware (e.g., network interface card and sound card). However, without manual annotations and configurations, it neither identifies device driver interfaces due to lack of kernel symbols nor meets conditions to initialize them.

SymDrive [41] is an S2E-based system to test Linux and FreeBSD drivers without devices, while overcoming the limitation of DDT. It uses a static analysis to auto-

matically identify driver code's key features such as entry point and loop, so, unlike DDT, it can correctly initialize device drivers without requiring manual effort. However, it also relies on source code instrumentation, so it cannot be applied to COTS OSes lacking debug information.

Trinity [24] and IOCTL Fuzzer [15] are system call fuzzers based on Linux and Windows, respectively. Before fuzzing a certain system call, they also try to construct pre-contexts, which is similar to CAB-FUZZ's on-the-fly technique. The key difference here is that CAB-FUZZ symbolizes the input, but these previous efforts randomly mutate input values only once. Thus, they have difficulties in detecting sophisticated conditions to trigger vulnerabilities.

Unlike the other systems described here, CAB-FUZZ does not rely on source code analysis or instrumentation, so it can be freely applied to COTS OSes. Furthermore, it does not suffer from the initialization problem thanks to its on-the-fly concolic testing.

Boundary Value Analysis. Several researchers have proposed boundary value analysis techniques [22, 23, 26, 38] to maximize branch coverage. For example, ADSE [22, 23] checks constraints at every path and loop and augments conditions to figure out which conditions generate maximum test cases. These approaches can detect the correct boundary conditions; however, the overall conditions will easily explode if we apply them to complex software, e.g., OSes. In contrast, CAB-FUZZ creates only two boundary states plus one arbitrary state for each symbolic array and loop such that it practically mitigates the state explosion problem.

Overflow Detection. CAB-FUZZ focuses on the boundaries of symbolic memories and loops because such boundaries could trigger stack or heap over/underflows. Several studies attempt to specialize symbolic execution to detect overflow and underflows. IntScope [47] and SmartFuzz [36] use symbolic execution to detect integer overflows. In addition, SmartFuzz covers integer underflows, narrowing conversions, and signed/unsigned conversions. Dowser [19] considers a buffer in a loop to detect its overflows and underflows. DIODE [44]'s goal is to find integer overflow errors at target memory locations. It uses a fine-grained dynamic taint analysis to identify all memory allocation sites, extracts target and branch constraints from instrumented execution, solves the constrains, and performs goal-directed conditional branch enforcement.

Although these methods work well, they rely on heavy static analysis and/or taint analysis to detect specific integers or buffers that could result in overflows. In contrast, CAB-FUZZ does not use such complicated analysis techniques when detecting boundaries, so it is more lightweight and practical than the previous techniques.

Lazy Initialization. CAB-FUZZ's on-the-fly concolic testing is a kind of lazy initialization technique [25, 49] that defers the initialization of memory or a data structure until it is actually used. Firmalice [43] is a binary analysis framework to analyze the firmware of embedded devices. It uses a lazy initialization technique to test memory because it does not know which code needs to be executed to initialize specific memory regions. When Firmalice detects a memory read from uninitialized memory during analysis, it pauses the execution and conducts the following procedures. First, it identifies other procedures that contain direct writes to the memory. Next, it labels the procedures as initialization procedures. Last, it duplicates the state: (1) resumes the execution without any modification to avoid possible crashes and (2) runs the initialization procedures before resuming the execution. However, a static program analysis is necessary to detect such initialization procedures.

UC-KLEE [14, 39, 40] directly tests individual functions instead of the whole program to improve scalability. To cope with missing pre-contexts of individual functions, it automatically generates symbolic inputs using lazy initialization. However, it still suffers from false positives due to invariants of data structures, state machines, and APIs, so it relies on manual annotations to reduce them.

On the contrary, CAB-FUZZ's on-the-fly concolic testing neither requires sophisticated static program analysis nor suffers from false positives. Also, it can be fully automated because it uses the real execution procedures of a target program.

8 Conclusion

In this paper, we presented a practical concolic testing tool, CAB-FUZZ, to analyze COTS OSes. CAB-FUZZ introduced two new memory symbolization techniques—synthetic symbolization and on-the-fly symbolization—allowing us to analyze COTS OSes without debug information and pre-contexts. It employed two boundary-state prioritization techniques: array- and loop-boundary prioritization, allowing us to prioritize potentially vulnerable paths. Evaluation results showed that CAB-FUZZ can detect 21 undisclosed unique crashes on Windows 7 and Windows Server 2008 while avoiding the state explosion problem.

Acknowledgements. We thank the anonymous reviewers and our shepherd, Mihai Christodorescu, for their helpful feedback. This research was supported by the NSF award DGE-1500084, CNS-1563848, CRI-1629851 ONR under grant N000141512162, DARPA TC program under contract No. DARPA FA8650-15-C-7556, DARPA XD3 program under contract No. DARPA HR0011-16-C-0059, and ETRI MSIP/IITP[B0101-15-0644].

References

[1] AVGERINOS, T., REBERT, A., CHA, S. K., AND BRUMLEY, D. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)* (Hyderabad, India, May–June 2014).

[2] BABIĆ, D., MARTIGNONI, L., MCCAMANT, S., AND SONG, D. Statically-Directed Dynamic Automated Test Generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)* (Toronto, Canada, July 2011).

[3] BOUNIMOVA, E., GODEFROID, P., AND MOLNAR, D. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)* (2013), pp. 122–131.

[4] BOYAPATI, C., KHURSHID, S., AND MARINOV, D. Korat: Automated Testing Based on Java Predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)* (2002).

[5] BUGRARA, S., AND ENGLER, D. Redundant State Detection for Dynamic Symbolic Execution. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)* (San Jose, CA, June 2013).

[6] CADAR, C., DUNBAR, D., AND ENGLER, D. 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)* (San Diego, CA, Dec. 2008).

[7] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)* (Alexandria, VA, Oct.–Nov. 2006).

[8] CADAR, C., AND SEN, K. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the ACM 56*, 2 (2013), 82–90.

[9] CHA, S. K., AVGERINOS, T., REBERT, A., AND BRUMLEY, D. Unleashing Mayhem on Binary Code. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)* (San Francisco, CA, May 2012).

[10] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2E: A Platform for In Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Newport Beach, CA, Mar. 2011).

[11] CVE. CVE-2013-5065, 2013. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-5065.

[12] DENG, X., LEE, J., AND ROBBY. Bogor/Kiasan: A k-bounded Symbolic Execution for Checking Strong Heap Properties of Open Systems. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2006).

[13] DOLAN-GAVITT, B., HULIN, P., KIRDA, E., LEEK, T., MAMBRETTI, A., ROBERTSON, W., ULRICH, F., AND WHELAN, R. LAVA: Large-scale Automated Vulnerability Addition. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)* (San Jose, CA, May 2016).

[14] ENGLER, D., AND DUNBAR, D. Under-Constrained Execution: Making Automatic Code Destruction Easy and Scalable. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)* (London, UK, July 2007).

[15] ESAGE LAB. IOCTL Fuzzer: Windows Kernel Driver Fuzzer. https://code.google.com/archive/p/ioctlfuzzer/.

[16] GODEFROID, P. Compositional Dynamic Test Generation. In *Proceedings of the 34th ACM Symposium on Principles of Programming Languages (POPL)* (Nice, France, Jan. 2007).

[17] GODEFROID, P., LEVEN, M. Y., AND MOLNAR, D. Automated Whitebox Fuzz Testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2008).

[18] GODEFROID, P., AND LUCHAUP, D. Automatic Partial Loop Summarization in Dynamic Test Generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)* (Toronto, Canada, July 2011).

[19] HALLER, I., SLOWINSKA, A., NEUGSCHWANDTNER, M., AND BOS, H. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Proceedings of the 22th USENIX Security Symposium (Security)* (Washington, DC, Aug. 2013).

[20] INTEL. Introduction to Intel Memory Protection Extensions. https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions.

[21] "J00RU" JURCZYK, M. Windows X86-64 System Call Table (NT/2000/XP/2003/Vista/2008/7/2012/8). http://j00ru.vexillium.org/ntapi_64/.

[22] JAMROZIK, K., FRASER, G., TILLMAN, N., AND DE HALLEUX, J. Generating Test Suites with Augmented Dynamic Symbolic Execution. In *Proceedings of International Conference on Tests and Proofs* (2013).

[23] JAMROZIK, K., FRASER, G., TILLMANN, N., AND HALLEUX, J. D. Augmented Dynamic Symbolic Execution. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2012).

[24] JONES, D. Trinity: A Linux System Call Fuzz Tester. http://codemonkey.org.uk/projects/trinity/.

[25] KHURSHID, S., PĂSĂREANU, C. S., AND VISSER, W. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2003).

[26] KOSMATOV, N., LEGEARD, B., PEUREUX, F., AND UTTING, M. Boundary Coverage Criteria for Test Generation from Formal Models. In *Proceedings of 15th International Symposium on Software Reliability Engineering (ISSRE)* (2004).

[27] KUZNETSOV, V., CHIPOUNOV, V., AND CANDEA, G. Testing Closed-Source Binary Device Drivers with DDT. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC)* (Boston, MA, June 2010).

[28] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, Colorado, Oct. 2014).

[29] MAJUMDAR, R., AND SEN, K. Hybrid Concolic Testing. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)* (Minneapolis, MN, May 2007).

[30] MARINESCU, P. D., AND CADAR, C. make test-zesti: A Symbolic Execution Solution for Improving Regression Testing. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)* (Zurich, Switzerland, June 2012).

[31] MARTIGNONI, L., MCCAMANT, S., POOSANKAM, P., SONG, D., AND MANIATIS, P. Path-Exploration Lifting: Hi-Fi Tests for Lo-Fi Emulators. In *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (London, UK, Mar. 2012).

[32] MICROSOFT. KeBugCheckEX routine (Windows Drivers). https://msdn.microsoft.com/en-us/library/windows/hardware/ff551961(v=vs.85).aspx.

[33] MICROSOFT. Methods for Accessing Data Buffers. https://msdn.microsoft.com/en-us/library/windows/hardware/ff554436(v=vs.85).aspx.

[34] MICROSOFT. NtCreateFile Function (Windows). https://msdn.microsoft.com/en-us/library/bb432380(v=vs.85).aspx.

[35] MICROSOFT. NtDeviceIoControlFile function (Windows). https://msdn.microsoft.com/en-us/library/ms648411(v=vs.85).aspx.

[36] MOLNAR, D., LI, X. C., AND WAGNER, D. A. Dynamic Test Generation To Find Integer Bugs in x86 Binary Linux Programs. In *Proceedings of the 18th USENIX Security Symposium (Security)* (Montreal, Canada, Aug. 2009).

[37] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M. K., AND ZDANCEWIC, S. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Dublin, Ireland, June 2009).

[38] PANDITA, R., XIE, T., TILLMANN, N., AND DE HALLEUX, J. Guided Test Generation for Coverage Criteria. In *Proceedings of IEEE International Conference on Software Maintenance (ICSM)* (2010).

[39] RAMOS, D., AND ENGLER, D. Practical, Low-effort Verification of Real Code using Under-constrained Execution. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)* (Snowbird, UT, July 2011).

[40] RAMOS, D. A., AND ENGLER, D. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *Proceedings of the 24th USENIX Security Symposium (Security)* (Washington, DC, Aug. 2015).

[41] RENZELMANN, M. J., KADAV, A., AND SWIFT, M. M. SymDrive: Testing Drivers without Devices. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Hollywood, CA, Oct. 2012).

[42] SAXENA, P., POOSANKAM, P., MCCAMANT, S., AND SONG, D. Loop-Extended Symbolic Execution on Binary Programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)* (Chicago, IL, July 2009).

[43] SHOSHITAISHVILI, Y., WANG, R., HAUSER, C., KRUEGEL, C., AND VIGNA, G. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2015).

[44] SIDIROGLOU-DOUSKOS, S., LAHTINEN, E., RITTENHOUSE, N., PISELLI, P., LONG, F., KIM, D., AND RINARD, M. Targeted Automatic Integer Overflow Discovery Using Goal-Directed Conditional Branch Enforcement. In *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Istanbul, Turkey, Mar. 2015).

[45] SLOWINSKA, A., STANCESCU, T., AND BOS, H. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2011).

[46] STEPHENS, N., GROSEN, J., SALLS, C., DUTCHER, A., WANG, R., CORBETTA, J., SHOSHITAISHVILI, Y., KRUEGEL, C., AND VIGNA, G. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2016).

[47] WANG, T., WEI, T., LIN, Z., AND ZOU, W. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2009).

[48] WANG, X., ZHANG, L., AND TANOFSKY, P. Experience Report: How is Dynamic Symbolic Execution Different from Manual Testing? A Study on KLEE. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)* (Baltimore, MD, July 2015).

[49] XIE, Y., AND AIKEN, A. Scalable Error Detection using Boolean Satisfiability. In *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages (POPL)* (Long Beach, CA, Jan. 2005).

[50] YANG, J., SAR, C., TWOHEY, P., CADAR, C., AND ENGLER, D. Automatically Generating Malicious Disks using Symbolic Execution. In *Proceedings of the 27th IEEE Symposium on Security and Privacy (Oakland)* (Oakland, CA, May 2006).