2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP) | 978-1-6654-3869-8/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICSE-SEIP52600.2021.00041

IntelliGen: Automatic Driver Synthesis for Fuzz Testing

Mingrui Zhang[1], Jianzhong Liu[2], Fuchen Ma[1], Huafeng Zhang[3], Yu Jiang*[1]
KLISS, BNRist, School of Software, Tsinghua Universiy[1]
ShanghaiTech University[2]
Huawei Technologies Co. Ltd, Beijing, China[3]

***Abstract*—Fuzzing is a technique widely used in vulnerability detection. The process usually involves writing effective fuzz driver programs, which, when done manually, can be extremely labor intensive. Previous attempts at automation leave much to be desired, in either degree of automation or quality of output.**

In this paper, we propose *IntelliGen*, a framework that constructs valid fuzz drivers automatically. First, *IntelliGen* determines a set of entry functions and evaluates their respective chance of exhibiting a vulnerability. Then, *IntelliGen* generates fuzz drivers for the entry functions through hierarchical parameter replacement and type inference. We implemented *IntelliGen* and evaluated its effectiveness on real-world programs selected from the Android Open-Source Project, Google's *fuzzer-test-suite* and industrial collaborators. *IntelliGen* covered on average 1.08×-2.03× more basic blocks and 1.36×-2.06× more paths over state-of-the-art fuzz driver synthesizers *FUDGE* and *FuzzGen*. *IntelliGen* performed on par with manually written drivers and found 10 more bugs.

***Index Terms*—Fuzz Testing, Fuzz Driver Synthesis, Software Analysis, Vulnerability Detection**

I. Introduction

Fuzzing is a popular software testing technique for vulnerability detection. It attempts to trigger bugs within the program by generating massive amounts of input and monitoring the program's runtime state. Fuzzers have been able to find numerous vulnerabilities within real-world applications and are of great interest in both academia and industry. AFL [1] and LibFuzzer [5] are two widely used fuzzers. Both use the classic genetic mutation algorithm to search for inputs that can improve coverage. Due to AFL's popularity, there have been much research to improve its efficiency. Notable examples include AFLFast [8], FairFuzz [17], MOpt [24], Angora [9] and Matryoshka [10].

Though fuzzing has achieved significant progress, there are still areas that require intensive manual labor, one of which is constructing effective fuzz driver programs for standalone libraries. This process usually requires the programmer to have a deep understanding of the program's source code, which is time-consuming and error-prone. Thus automating this process is imperative to improving the effectiveness of fuzzing.

There have been some work into automated fuzz driver generation. Google recently proposed a framework *FUDGE* [6] to generate fuzz drivers semi-automatically. *FUDGE* constructs fuzz drivers by scanning the program's source code to find vulnerable function calls and generates fuzz drivers using parameter replacement. It works well in specific scenarios but tends to generate excessive fuzz drivers for large projects which require manual removal of invalid results. Furthermore, it is infeasible to try all candidate drivers. Another example would be *FuzzGen* [14], which leverages a whole system analysis to infer the library's interface and synthesize fuzz drivers based on existing test cases accordingly. Its performance relies heavily upon the quality of existing test cases.

In production environments, we face two significant challenges when constructing fuzz drivers automatically: (1) One is to locate high-value entry functions. A high-value entry function should have the ability to reach lower levels of the program, yielding more code coverage after running for an extensive period. In addition, an entry function that contains vulnerable operations such as *memcpy()* is considered to be more vulnerable and should require more attention. (2) The other is to synthesize a valid and effective fuzz driver based on the identified entry function. A correct fuzz driver should be able to call the target function with suitable parameters.

In this paper, we present *IntelliGen* to address these challenges and construct fuzz drivers automatically. *IntelliGen* works as follows. First, it scans the target program, looking for functions with high vulnerability priority as potential entry functions. Then, it synthesizes parameters using algorithmic methods. Finally, *IntelliGen* constructs the fuzz driver which calls the entry function with Address-Sanitizer [28] enabled to initialize the fuzzing process. To evaluate its effectiveness, we test *IntelliGen* on some real-world projects selected from Android Open-Soure Project, Google's *fuzzer-test-suite* and industrial collaborators. The results show that *IntelliGen* covers 1.08×-2.03× more basic blocks, 1.36×-2.06× more paths, and detects 10 more bugs than the fuzz driver synthesizer *FUDGE* and *FuzzGen* in total. Compared with the drivers written manually by domain experts, *IntelliGen* covers almost the same number of branches, paths, and bugs.

In summary, we make the following contributions:

- We propose a new fuzz driver synthesis framework, *IntelliGen*, which locates vulnerable functions and constructs fuzz drivers automatically, reducing the total amount of manual intervention required.

*Yu Jiang is the corresponding author. This research is sponsored in part by the NSFC Program (No. 62022046, U1911401, 61802223), National Key Research and Development Project (Grant No. 2019YFB1706200), the Huawei-Tsinghua Trustworthy Research Project (No. 20192000794).

978-1-6654-3869-8/21/$31.00 ©2021 IEEE
DOI 10.1109/ICSE-SEIP52600.2021.00041

- We implement *IntelliGen* using the LLVM framework with generalized entry function location and accurate parameter inference, allowing for increased performance and compatibility than the state-of-the-art.
- We test *IntelliGen* on real-world projects selected from the Android Open-Soure Project, Google's *fuzzer-test-suite* and projects from industrial collaborators. The result shows that *IntelliGen* covers 1.08×-2.03× more blocks, 1.36×-2.06× more paths and detects 10 more bugs than *FUDGE* and *FuzzGen* and performs at least as well as manually written drivers.

The rest of this paper is organized as follows. Section II introduces related work. Section III explains *IntelliGen*'s design, including the *Entry Function Locator* and the *Fuzz Driver Synthesizer*. Section IV covers the implementation details. Section V evaluates *IntelliGen* on real-world programs and compares its results against *FUDGE* and *FuzzGen*. Section VI demonstrates the application in real industrial practices. Section VII discusses the problems we encountered and potential future work. We conclude in Section VIII.

II. Related Work

In this section, we introduce related work on fuzzing. We mainly discuss fuzzing in industry, automatic fuzz driver generation, API usage analysis and unit test generation.

(1) Fuzzing in industry. Fuzzing is a powerful technique for detecting vulnerabilities. There have been much research effort in improving fuzzing performance. InteFuzz [18], DeepFuzz [20] and SAFL [31] improves the overall efficiency of fuzzing algorithms. Zeror [33] increases the fuzzing throughput by optimizing the fuzz target's execution speed. EnFuzz [11] and PAFL [19] allows fuzz engines to scale better. The effectiveness of the aforementioned projects have been demonstrated in numerous industrial applications [12], [13], [21]–[23], [29].

Some other work have been conducted to streamline the fuzzing process. For instance, Google's OSS-Fuzz [3], which uses libFuzzer [5] and AFL [1] as its backend, has found thousands of bugs over a period of 5 months. ClusterFuzz [2] is the distributed infrastructure behind OSS-Fuzz, which automatically builds and executes binaries with different versions and finds the version that introduces a specific bug. AFL targets the entire executable, generates random inputs for the program and monitors its runtime state. LibFuzzer fuzzes library functions by interfacing through the function *LLVMFuzzerTestOneInput()*. It generates random data for this function and checks whether the program crashes during execution. Research based on LibFuzzer has led to a number of improved implementations, such as HonggFuzz [4].

(2) Automatic fuzz driver generation. Fuzz drivers are required when one wishes to fuzz a library function. Originally, testers wrote fuzz drivers for the target library by hand, which is inefficient and error-prone. To synthesize fuzz drivers automatically, Google proposes a framework named *FUDGE* [6]. It scans the source code of the project for the vulnerable API, synthesizes the interface function *LLVMFuzzerTestOneInput()* which calls the vulnerable API, and fuzzes it using LibFuzzer [5]. *FUDGE* operates by finding a function A, which calls another function B with the signature *(uint8_t*, size_t)*. Then it considers function A as the entry function and binds the buffer generated by the fuzz engine with function B's signature. This process will generate a large number of candidate drivers. *FUDGE* shows them to testers directly, allowing them to modify those drivers manually to guarantee correctness. *FuzzGen* [14] scans the source code of the target project, finds the dependency of each API function, and generates interfaces to call the API functions based on existing test cases. Without qualified test cases, it will fail to generate correct drivers.

(3) API Usage Mining. Similar to API usage mining, fuzz driver synthesis needs to identify meaningful entry functions. MAPO [32] searches and mines for the most frequent called APIs in the target project. GrouMiner [26] mines usage patterns based on the graph of the project. APIExample [30] extracts usage examples with similar functions.

(4) Unit-test generation. Unit-test generation is another close research area to fuzz driver synthesis. Kampmann et al. [16] presents a method to automatically extract parameterized unit tests from system test executions. Testful [7] generates test cases for Java classes and methods. Pacheco et al. [27] presents a technique that improves random test generation by incorporating feedback obtained from executing test inputs as they are created. GRT [25] uses static and dynamic analysis to include information on program types, data, and dependencies in various stages of automated test generation. GenRed [15] generates and reduces object-oriented test cases.

III. *IntelliGen* Design

Figure 1 shows the high-level architecture of *IntelliGen*'s design. *IntelliGen* consists of two main modules: *Entry Function Locator* and *Fuzz Driver Synthesizer*. *Entry Function Locator* locates and sorts the vulnerable functions that contain many vulnerable operations such as *memcpy()* in the target project. *Fuzz Driver Synthesizer* synthesizes arguments for the entry functions located by *Entry Function Locator* and ensures there will be no memory safety issues regarding the entry function arguments.

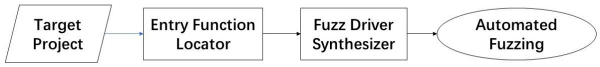

Fig. 1: Overview of *IntelliGen*. It uses the *Entry Function Locator* to identify potential functions to fuzz and leverages the *Fuzz Driver Synthesizer* to construct fuzz drivers.

A. Entry Function Locator

The effectiveness of fuzz testing relies upon the quality of the entry function of the driver program. An effective entry function may in turn call numerous other functions, potentially

leading to higher edge coverage. In contrast, an ineffective driver may perform many checks on the supplied arguments, reducing the overall efficiency of fuzz testing A powerful fuzz driver should allow us to bypass those checks and reach the actual program logic directly.

To Fuzz a project, testers need to communicate with its developers or read the relevant documents to manually find potential entry functions and construct valid function calls. This cannot guarantee quality drivers and is highly time consuming. To reduce manual labor, *FUDGE* and *FuzzGen* have been developed to synthesize fuzz drivers automatically. *FUDGE* looks for functions with parameters *(uint8_t*, size_t)* and assigns them with the buffer produced by fuzz engine. *FuzzGen* extracts API function dependency from the test cases and constructs a function to call API functions in a special order. However, *FUDGE* may produce many ineffective drivers thus requiring manual intervention to pick and update effective fuzz drivers. *FuzzGen* on the other hand depends on the quality of the project's test cases. Without them, *FuzzGen* is incapable of generating any fuzz drivers.

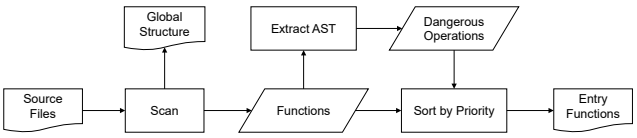

Fig. 2: *IntelliGen*'s process for locating entry functions.

To solve these problems and automate the driver generation process, *IntelliGen* considers all functions as potential entry functions, evaluates their vulnerability priority and selects those with the highest priority as entry functions. Figure 2 gives an overview of *IntelliGen*'s *Entry Function Locator*.

First, *IntelliGen* scans the project and extracts all functions found. To acquire the most suitable entry functions, *IntelliGen* calculates the priority of each function based on the number of memory dereferencing operations.

Algorithm 1 presents our method of accessing the priority for each function. It takes the list of all source files *fileList* as its argument and returns *funcMap*, a map between the functions and their priorities. The algorithm operates by scanning all the input files. When it encounters a new function, it first initializes its priority to 0 and inserts it into *funcMap*, as shown in line 2-7. The algorithm then scans all statements in the function to get its priority, as shown in line 8-17. We regard the following types of statements as potential vulnerable statements:

- **Dereferencing a pointer**. Buffer overflows are the most common security issues in industrial situations. These can only be triggered by dereferencing an invalid pointer.
- **Calling a memory related function**. Some *libc* library functions such as *memset()* and *memcpy()* may dereference pointers internally. These functions are a potential source of buffer overflows.

ALGORITHM 1: Algorithm for evaluating potential entry functions' priorities.

```
   Input: source code files fileList
   Output: functions with their priority funcMap
1  funcMap = Map<Function, Priority>
2  foreach file in fileList do
3      foreach func in file do
4          if func in funcMap then
5              continue
6          end
7          funcMap.push(func, 0)
8          priority = 0
9          foreach stat in func do
10             if stat dereferences a pointer then
11                 priority += 1
12             else if stat processes memory then
13                 priority += 1
14             else if stat calls func2 then
15                 priority += func2.getPriority()
16             end
17         end
18         funcMap.update(func, priority)
19     end
20 end
21 return funcMap
```

- **Calling other functions in the same project.** If the child function can potentially cause buffer overflows, then the parent function should also be considered as vulnerable.

A function that contains a greater amount of vulnerable statements should receive a higher priority. After processing all statements in the function, its priority value in *funcMap* is updated accordingly, as shown in line 18. Finally, the algorithm returns *funcMap*, the priorities of all functions, as shown in line 21.

Then, *IntelliGen* selects the functions with the highest priority as the entry functions. The algorithm of the *Entry Function Locator* is presented in Algorithm 2. It takes two parameters, namely *maxNumber*, specifying the max number of entry functions, *fileList*, containing the target program's source files, and returns *funcList* as entry functions. First, *Entry Function Locator* evaluates all functions in the *fileList* by their respective vulnerable priorities, as shown in lines 1-6. Then, it sorts the functions by their priorities, as shown in line 7. Next, it retrieves the *maxNumber*-highest-priority functions as the final entry functions, as shown in lines 8-14.

Compared with *FUDGE*'s pattern matching and *FuzzGen*'s API function searching in existing test cases, *IntelliGen*'s design is guarenteed to be general and versatile. *IntelliGen* can accurately identify vulnerable functions without resorting to additional information.

B. Fuzz Driver Synthesizer

The effectiveness of fuzz testing is highly dependent upon whether the driver is capable of invoking the entry function correctly. An optimal driver can transform the input generated by the fuzz engine to the target function's arguments correctly. Conversely, a faulty driver may even introduce errors, seriously

ALGORITHM 2: Algorithm of *Entry Function Locator*

Input: maximum number of entry functions *maxNumber*
source code files *fileList*
Output: entry functions *funcList*

```
funcMap = Map<Function, Priority>
foreach file in fileList do
    foreach func in file do
        funcMap.push(func.getPriority(func))
    end
end
funcMap = funcMap.sortBy(Priority)
funcList = []
while maxNumber > 0 and not funcMap.empty() do
    firstFunc = funcMap.pop()
    funcList.push(firstFunc)
    maxNumber -= 1
end
return FuncList
```

affecting the performance of fuzzing. Therefore, synthesizing high-quality fuzz drivers based on the located entry functions is vital towards good fuzzing performance.

Fuzz drivers are tightly coupled with the fuzzing engines. *LibFuzzer* is a widely-used fuzzing engine and has been applied to find many serious bugs in previous evaluations. Therefore, we choose *LibFuzzer* as *IntelliGen*'s fuzzing engine. *LibFuzzer* uses the interface function *LLVMFuzzerTestOneInput()* to initiate its fuzzing process, so *IntelliGen* synthesizes this interface function for the located entry function.

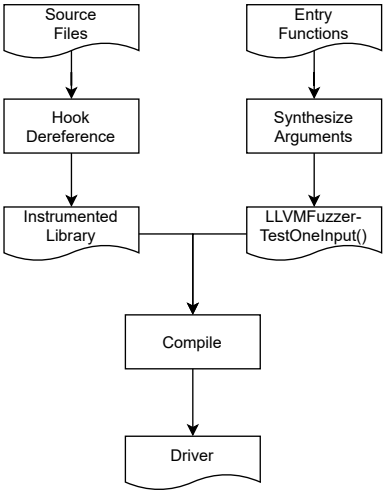

Fig. 3: *Fuzz Driver Synthesizer*'s process of fuzz driver synthesis.

Figure 3 shows how *IntelliGen* synthesizes a fuzz driver in detail. It takes the entry function as its input, and returns a fuzz driver in the form of the *libFuzzer* interface function *LLVMFuzzerTestOneInput()*. First, *IntelliGen* builds an empty function *LLVMFuzzerTestOneInput()*. Then, *IntelliGen* instruments the entry function's arguments to assign valid values at runtime. Specifically, it analyzes the argument's value type using the following rules:

ALGORITHM 3: Runtime Value Assignment Routines. Functions *getScalarValue()* and *getPointerValue()* are inserted to assign values for scalar and pointer objects. Functions *LoadInstHook()* and *StoreInstHook()* are inserted before each load and store instruction.

```
Function getScalarValue(size):
    return readFromBuffer(size);
endFunction
Function getPointerValue():
    ptr = malloc(SIZE);
    markAsUnassigned(ptr, SIZE);
    return ptr;
endFunction
Function LoadInstHook(ptr):
    if NotAssigned(ptr) then
        /* Compile-time determined        */
        if isScalarType then
            *ptr = getScalarValue(OBJECT_SIZE);
        else if isPointerType then
            *ptr = getPointerValue();
        else if isStructOrArrayType then
            recursively assign all members ;
        end
        markAsAssigned(ptr);
    end
endFunction
Function StoreInstHook(ptr):
    markAsAssigned(ptr);
endFunction
```

- **Scalar Types**: For scalar types, such as integers and floating point numbers, *IntelliGen* instruments runtime instructions that read subsequent bytes from the input buffer generated by *LibFuzzer* and assign them to the argument value.
- **Pointer Types**: For pointer types, *IntelliGen* instruments a runtime function that allocates memory without initialization and assigns the pointer value with its starting address. The size of the allocated memory should be large enough to accommodate either the object that the pointer points to or a small integer array.
- **Array or Structure Types**: For array or structure types, *IntelliGen* assigns values for its member values recursively by repeating this process on each member.

Next, *IntelliGen* scans the entry function and inserts statements to *lazy-stores* values into arguments with pointer type. Generally, we can not know the actual type of a pointer until it is dereferenced, because C allows casting any pointer to *void** and vice versa. Hence, we can not assign a fitting value to the pointer based on its current type. To solve this, we must know the type in which it is used, and one method to acquire this information is to trace dereference operations of the pointer. Therefore, we delay the assignment of the values referenced by pointers until its first usage, hence the name *lazy-store*. *IntelliGen* keeps track of whether a memory range has been previously assigned a value. All pointer values in the entry function's arguments are initially marked as *unassigned*. For store operations, *IntelliGen* inserts instructions

that mark the memory range as *assigned*. For load instructions, *IntelliGen* instruments a runtime function that checks if the memory range is marked as *assigned*. If not, it assigns a value corresponding to the dereferenced type using the method for entry function arguments. When we find a *load* operation and the memory that it points to has not been previously assigned, *IntelliGen* constructs a parameter with the same type using the method for constructing argument values and stores it into the relevant memory area just before loading from it. We also expand memory related functions such as *memcpy()* and *memset()* into load and store operations for a more complete analysis. Algorithm 3 shows the instrumented functions that assign values in the situations described above during runtime.

IV. Implementation

In this section, we introduce the implementation details of *IntelliGen*. We implement *IntelliGen* using the LLVM compiler framework. The interface function *LLVMFuzzerTestOneInput()* generated by *IntelliGen* is constructed at the Intermediate Representation (IR) level. *IntelliGen* links this function with the library code together into one bitcode file. We can then compile the bitcode file to an executable fuzz driver. We mainly solve the following challenges during implementation:

Evaluating the priority of functions. When locating entry functions, we need to calculate their respective priorities. In essence, we should count the number of statements which dereferences a pointer or calls other functions to processes memory. LLVM-IR uses *load* and *store* instructions to read from and write to memory through a pointer respectively and *call* instructions to call a function, so we can trace these three kinds of instructions to get the priority for any function.

Choosing effective entry functions. *IntelliGen*'s *Entry Function Locator* can locate a few potentially effective entry functions, but the user should have to choice to select which function to fuzz. *IntelliGen* shows the recommended entry functions and testers can manually intervene to determine which entry functions to use or generate drivers for all potential entry functions automatically.

Avoiding redundant memory assignments. *IntelliGen* will only lazy-store into any memory area if the area has not been previously marked as occupied. To keep track of all occupied memory segments, *IntelliGen* maintains a global map and inserts an instruction to mark the corresponding memory as filled before a corresponding store instruction.

Synthesizing complex arguments for the entry function. An entry function may contain pointer-typed parameters, each pointing to a structure containing another complex structure. By using the lazy-store technique, *IntelliGen* can generate these arguments correctly. For any pointer arguments, as mentioned in Algorithm 3, *IntelliGen* assigns a plain chunk of memory. *IntelliGen* will only store an object if it is ever dereferenced. For all member variables, they are generated recursively. This allows *IntelliGen* to handle parameter construction with ease.

Assigning appropriate values for arguments. Generating random values for an argument may decrease the effectiveness of fuzz testing. To find an appropriate value for an argument, *IntelliGen* scans the IR of the function in search for comparison instructions. If one of the operands used by a comparison instruction is an argument pending assignment, then the other operand will be considered as an appropriate value for the argument. *IntelliGen* will generate additional code to decide if the appropriate value should be assigned to the argument.

Filtering out useless drivers. Though *IntelliGen* uses many techniques to generate arguments for the entry function, We cannot guarantee that all synthesized drivers will be valid, especially for entry functions containing function pointer arguments. To evaluate a synthesized driver's effectiveness, we execute the driver for a short amount of time automatically and monitor its runtime state. If it crashes, we consider the driver to be invalid and it will be removed. An invalid driver is often caused by dereferencing a function pointer or an assertion statement failing. Memory leaks may also invalidate a fuzz driver. If an entry function allocates memory on the heap but does not free it, then a memory leak error will occur. To avoid memory leaks, *IntelliGen* hooks memory allocation/deallocation functions such as *malloc()* and *free()* with *IntelliGen_alloc()* and *IntelliGen_free()*. *IntelliGen_alloc()* records the memory it allocates, and *IntelliGen_free()* remove the record of the memory it frees. At the end of function *LLVMFuzzerTestOneInput()*, all memory recorded by *IntelliGen_alloc()* but not freed by *IntelliGen_free()* will be freed altogether.

V. Evaluation

To examine the effectiveness of *IntelliGen*, we evaluate it on the 6 real-world libraries of Android Open-Soure Project (AOSP) used in the evaluation of *FuzzGen*, 9 real-world projects of Google's *fuzzer-test-suite* with manually written drivers for the comparison of Google's FUDGE, and 3 real-world projects from industrial collaborators. These projects consist of image processing libraries (*libjpeg*), file processing libraries (*libxml2*, *JSON*), regular expression engines (*pcre2*), asynchronous resolver libraries (*c_ares*), font compression and decompression libraries (*woff2*, *libhevc*, *libhavc*), and font shaping libraries (*harfbuzz*).

We use two commonly used metrics to evaluate the effectiveness of automatic generated fuzz drivers on these real-world libraries, namely basic block coverage and path coverage. An effective driver should be capable of allowing the fuzz engine to cover a significant amount of the code. We collect the code coverage information using LLVM-cov, which provides us information on block coverage, and path coverage.

We conduct our experiments on a machine with Intel Xeon Gold 6148 processors and 128GiB of memory running on 64-bit Ubuntu Linux 18.04. We run each fuzz driver 10 times, each using four threads over a period of 6 hours and report the average coverage statistics.

A. Comparison with FuzzGen

We compare *IntelliGen* and *FuzzGen* on the effectiveness of synthesized drivers based on the six projects used in the evaluation of *FuzzGen*. For *FuzzGen*, we use the same fuzz driver provided with their project[1]. For *IntelliGen*, we use *IntelliGen*'s *Entry Function Locator* to identify an initial set of entry functions, and utilize *Fuzz Driver Synthesizer* to synthesize a fuzz driver that calls the identified entry functions. The results are presented in Table I, which shows the number of blocks, and paths.

TABLE I: Number of blocks and paths covered by fuzz drivers synthesized by *IntelliGen* and *FuzzGen*. We do not list the bugs found in these libraries since these do not have a standard bug list.

Project		Blocks	Paths
libavc	*IntelliGen*	999	765
	FuzzGen	579	81
libgsm	*IntelliGen*	1258	659
	FuzzGen	1339	1173
libhevc	*IntelliGen*	928	476
	FuzzGen	558	188
libmpeg2	*IntelliGen*	1105	630
	FuzzGen	44	70
libopus	*IntelliGen*	124	112
	FuzzGen	-	-
libvpx	*IntelliGen*	3221	924
	FuzzGen	-	-
total	*IntelliGen*	7635	3566
	FuzzGen	2520	1512
	increase	2.03X↑	1.36X↑

As shown in Table I, *IntelliGen* is able to achieve better code coverage than *FuzzGen* on most projects, exceeding 50% more block and path coverage on *libavc* and *libhevc* over *FuzzGen*. The only exception is *libgsm*, where *IntelliGen* covers slightly less blocks than *FuzzGen* though covering more paths. The fuzz drivers provided by *FuzzGen* for *libopus* and *libvpx* are invalid, whereas *IntelliGen* can synthesize correct drivers for these two libraries. Figure 4 demonstrates the effectiveness of the fuzz drivers synthesized by *IntelliGen* and *FuzzGen* over time.

We examined the source code of the libraries we tested and analyzed the reason for *IntelliGen*'s better performance. In the *libavc* and *libhevc* libraries, *IntelliGen* and *FuzzGen* both synthesized fuzz drivers for the high level APIs (*ih264d_api_function()* and *ihevcd_cxa_api_function()*. However, as *IntelliGen* is capable of constructing argument values using principled methods while *FuzzGen* is only capable of inferring possible values from test cases, the driver *IntelliGen* is more versatile and potentially capable of reaching more code. A more prominent example is *libmpeg2*. *IntelliGen* and *FuzzGen* are both capable of synthesizing fuzz drivers for the API function *impeg2d_api_function()*. However, as the project does not provide test cases with much insight, *FuzzGen* cannot construct arguments that allow the API function to reach large proportions of the code, thus limiting the fuzz driver's performance. *IntelliGen*'s approach on the other hand allows for vastly better performance.

Both *IntelliGen* and *FuzzGen* synthesized fuzz drivers on a single exposed API function for the previous libraries. In *libgsm* however, there are multiple potential entry functions. *IntelliGen*'s *Entry Function Locator* identifies four entry functions and its *Fuzz Driver Synthesizer* constructs one driver each and one driver which calls all four functions consecutively, thus five drivers in total. *FuzzGen* also provides five drivers. Surprisingly, the best performing drivers for both *IntelliGen* and *FuzzGen* call the same function only.

Overall, *IntelliGen* performs better on the majority of libraries than *FuzzGen*, even though *FuzzGen* leverages additional information extracted from test cases while *IntelliGen* relies only on algorithmic parameter synthesis.

B. Comparison with FUDGE

We also compare the performance of *IntelliGen* and *FUDGE* regarding the effectiveness of synthesized drivers. We select the driver from *FUDGE* with the highest performance out of the 100 sampled valid drivers. The results are presented in Table II , which show the number of blocks, the number of paths, and unique crashes of each driver respectively.

TABLE II: Number of blocks, paths covered and bugs found by *IntelliGen* and *FUDGE* on Google's *fuzzer-test-suite*.

Project		Blocks	Paths	Bugs
re2	*IntelliGen*	2050	11228	0
	FUDGE	1203	100	0
harfbuzz	*IntelliGen*	6259	12708	1
	FUDGE	82	3952	0
guetzli	*IntelliGen*	692	7728	3
	FUDGE	80	1460	0
libjpeg	*IntelliGen*	1236	780	0
	FUDGE	1059	738	0
woff2	*IntelliGen*	100	3936	2
	FUDGE	69	59	0
json	*IntelliGen*	903	3278	1
	FUDGE	546	632	1
libxml	*IntelliGen*	2355	128	1
	FUDGE	1529	12	0
pcre2	*IntelliGen*	14852	21273	22
	FUDGE	9144	13020	21
c_ares	*IntelliGen*	31	67	2
	FUDGE	–	–	–
total	*IntelliGen*	28478	61126	32
	FUDGE	13712	19973	22
	increase	1.08X↑	2.06X↑	0.45X↑

The third and fourth column of Table II presents the block coverage and path coverage statistics for each project. To obtain the total block coverage, we re-run each seed and merge all the blocks together. To obtain the total path coverage, we add the number of paths of each driver, since different drivers with different entry functions do not share the same path.

[1] *libopus* and *libvpx*'s sample drivers provided by *FuzzGen* are invalid and we could not execute them successfully.

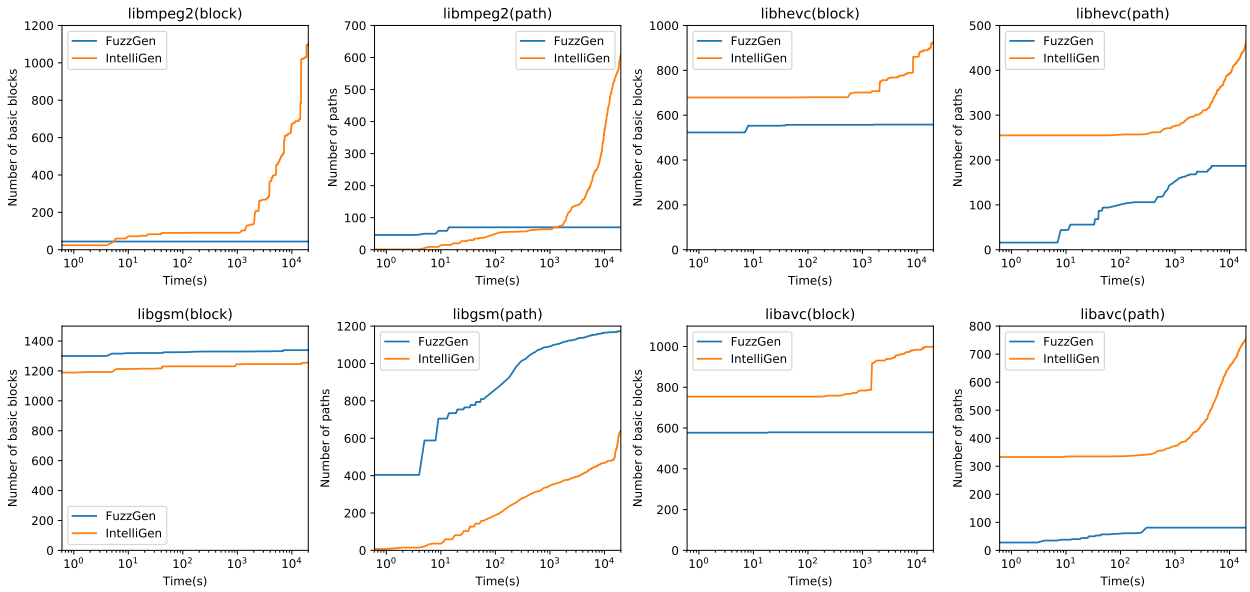

Fig. 4: Comparisons of the number of blocks and paths covered in *libavc*, *libgsm*, *libhevc* and *libmpeg2* by *IntelliGen* and *FuzzGen*. The drivers synthesized by *FuzzGen* for *libopus* and *libvpx* are invalid and not demonstrated in this figure.

We can observe that for each project, *IntelliGen* covers significantly more blocks than *FUDGE*. *IntelliGen* also covers much more paths than *FUDGE* on most projects except *libjpeg*, since we skip some similar high-level entry functions. Overall, *IntelliGen*'s block coverage outperforms that of *FUDGE* by 79.54%, while *IntelliGen* covers 105.52% more paths than *FUDGE*. On projects such as *harfbuzz*, *guetzli*, *woff2* and *json*, *IntelliGen* is able to cover at least 100% more blocks and paths than *FUDGE*. This is because *IntelliGen* tends to find high-level functions as potential entry functions, which usually call a series of low-level functions. However, *FUDGE* only finds functions which call another function with the signature *(const uint8_t*, size_t)*. These functions may be low-level functions and thus they are unable to cover much of the code. The last two rows of Table II show that on project *c-ares*, *FUDGE* fails to synthesize any valid fuzz drivers. Therefore no function in *c-ares* calls another function with the target signature. However, *IntelliGen* can directly synthesize parameters for the entry function, which makes *IntelliGen* more versatile than *FUDGE*.

The fifth column shows the unique bugs each driver detects. The fuzz drivers synthesized by *IntelliGen* trigger bugs in seven projects, while those synthesized by *FUDGE* trigger bugs in only two projects. In total, *IntelliGen* finds 32 potential bugs, but *FUDGE* only finds 22 potential bugs on *json* and *pcre2*, since *IntelliGen* tends to use high-level and more vulnerable functions as its entry functions, thus it is more likely to trigger potential bugs.

Based on the results shown in Table II, we can conclude that *IntelliGen* locates more effective entry functions, achieving higher block and path coverage and more unique bugs detected than *FUDGE*, without the need for manual selection or modification.

VI. Case Study on Real Projects

In this section, we use many real-world projects from Google's *fuzzer-test-suite* and our industrial collaborators to demonstrate the effectiveness of *IntelliGen*.

1) Real-world Project in Google's fuzzer-test-suite: We first dissect the manually written driver and the fuzz drivers generated by *IntelliGen* and *FUDGE* for the *pcre2* library.

(1) The original manually written driver. As presented in Listing 1, the driver written by domain experts invokes three entry functions: *regcomp()*, *regexec()* and *regfree()*. First, it calls *regcomp()* on the buffer generated by the fuzz engine. Then, it calls *regexec()* with the previous variable *preg*. Finally, it calls *regfree()* to free the variable *preg*.

(2) The driver synthesized by *IntelliGen*. *IntelliGen* locates *pcre2_match()* as one of the functions with the highest priority and regards it as the entry function for driver synthesis, whose function prototype is shown in Listing 2.

To synthesize fuzz drivers for the entry function *pcre2_match()*, *IntelliGen* needs to synthesize its parameters. First, *IntelliGen* regards the second and third parameters as the buffer and its size, then binds them with the buffer generated by the fuzz engine. For the three complex parameters *code*, *match_data* and *mcontext* in *pcre2_match()*, *IntelliGen* calls function *pcre2_compile()* to get the variable

Listing 1 Driver written by domain experts

```
extern "C" int LLVMFuzzerTestOneInput(const unsigned char *data,
        size_t size)
{
    if (size < 1) return 0;
    regex_t preg;
    string str( reinterpret_cast <const unsigned char*>(data),
          size);
    string pat(str);
    int flags = data[size/2] - 'a';
    if (0 == regcomp(&preg, pat.c_str(), flags))
    {
        regmatch_t pm[5];
        regexec(&preg, str.c_str(), 5, pm, 0);
        regfree(&preg);
    }
    return 0;
}
```

Listing 2 Function pcre2_match

```
PCRE2_EXP_DEFN int PCRE2_CALL_CONVENTION pcre2_match
    (const pcre2_code *code, PCRE2_SPTR subject, PCRE2_SIZE
    length, PCRE2_SIZE start_offset, uint32_t options,
    pcre2_match_data *match_data, pcre2_match_context *
    mcontext);
```

code, calls *pcre2_match_data_create_from_pattern()* to get the variable *match_data*. As for *mcontext*, *IntelliGen* does not find a proper function to initialize it, so it synthesizes a variable with the same type and assigns all bits with 0. In reality, the variable is a *NULL* pointer. As a result, *IntelliGen* synthesizes a candidate fuzz driver for function *pcre2_match()*. We run the driver automatically and find that it results in a memory leak. To fix the memory leak, *IntelliGen* calls function *pcre2_match_data_free()* to free the variable *match_data*, and calls *pcre2_code_free()* to free the variable *code*.

Finally, *IntelliGen* synthesizes a valid fuzz driver with the two free functions, as presented in Listing 3. The driver first calls *pcre2_compile()* to obtain a pointer of type *pcre2_code()*. Then it checks whether the `pointer` is *NULL*. If not, *pcre2_match_data_create_from_pattern()* is called to get a pointer of type *pcre2_match_data()*. The driver also checks whether the acquired pointer is *NULL*. If not, the driver calls the entry function *pcre2_match()* with the generated arguments to initiate fuzzing. Finally, it calls *pcre2_match_data_free()* and *pcre2_code_free()* to free the two pointers previously generated.

(3) The driver synthesized by FUDGE. We also utilize *FUDGE* to synthesize fuzz drivers. *FuzzGen* can not generate valid driver without providing additional test cases. First, *FUDGE* finds a function *compile_pattern()* which calls another function *pcre2_compile()* with the function signature *(const uint8_t*, size_t)*. Then, *FUDGE* extracts *pcre2_compile()*'s relevant code snippets in *compile_pattern()* and generates a driver. Since *FUDGE* does not propose a method to assign value for pointer parameters, we need to

Listing 3 Driver synthesized by *IntelliGen*

```
uint32_t p1, p2, p5;
uint64_t p3, p4;
pcre2_code* v1 = pcre2_compile(data, size, p1, &p2, &p3, NULL)
    ;
if (v1)
{
    pcre2_match_data *v2 =
        pcre2_match_data_create_from_pattern(v1, NULL);
    if (v2)
    {
        pcre2_match(v1, data, size, p4, p5, v2, NULL);
    }
    pcre2_match_data_free(v2);
}
pcre2_code_free(v1);
```

manually assign all pointer parameters to *NULL*. In addition, this driver will cause a memory leak similar to the one found in *IntelliGen*. Since *FUDGE* does not propose a way to free allocated memory correctly, we have to call *pcre2_code_free()* manually. The final modified driver is shown in Listing 4.

Listing 4 Driver synthesized by FUDGE with modification

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data,
        size_t size) {
    unsigned char buffer[8292] = {};
    memcpy(buffer, data, size);
    PCRE2_SIZE erroffset;
    int errcode;
    pcre2_compile_context
        *compile_context = NULL;
    int options = data[0];
    pcre2_code * v1 = NULL;
    if (v1 != NULL)
        return 0;
    v1 = pcre2_compile(buffer, -1, options, &errcode, &erroffset
        , compile_context);
    pcre2_code_free(v1);
    return 0;
}
```

(4) Code coverage of different drivers. We run each driver for 24 hours, repeat 10 times and collect their average code coverage information as presented in Figure 5.

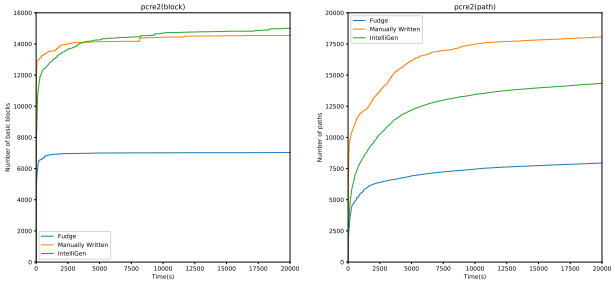

Fig. 5: Block and path coverage for drivers written manually, and drivers synthesized by *FUDGE* and *IntelliGen*.

We can observe that the block coverage grows fast initially for all drivers, then the growth changes to an approximate logarithmic rate. This is because the fuzzer tends to discover 'easy' blocks rapidly, but once they have been mostly discovered, the fuzzer will require relatively more effort to cover the 'hard' blocks. Also, there is a theoretical upper limit for the coverage of each driver: an inefficient driver that calls a low-level API function can not cover many blocks, while an efficient driver that calls a high-level API function can cover a lot of blocks after running a significant amount of time. The path coverage trend is similar to that of the block coverage but continues to grow even after fuzzing for 24 hours. This is because while the seeds may cover the same set of blocks, the order in which they are visited constitute different paths.

We can also see that *IntelliGen* and *FUDGE* both cover fewer blocks than the original driver written by domain experts in the first few seconds. But after fuzzing for an substantial period, the driver synthesized by *IntelliGen* covers an increasing amount of blocks and can even cover a few more blocks than the manually written driver at the end of 24 hours. The driver synthesized by FUDGE, however, does not exhibit an obvious coverage growth after fuzzing for 24 hours. This is because the driver synthesized by FUDGE does not contain the core function *pcre2_match()*. If we manually modify the driver to call *pcre2_match()* or *regexec()*, then it can cover more blocks after a substantial fuzzing period.

2) Real-world projects from industrial collaborator: We use *IntelliGen* on three real-world projects used by Huawei Technologies: *mxml v2.9*, *mxml v2.12* and *libevent*. These three projects are also widely used in other industrial communities. The overall results are presented in Table III, where *FuzzGen* did not generate valid drivers because of the absence of test cases. The first, second and third column represents the number of blocks covered in each project, the number of paths covered and the number of unique bugs detected, respectively.

TABLE III: Number of blocks, paths, and unique bugs

Project		Blocks	Paths	Unique bugs
mxml v2.9	*IntelliGen*	715	503	5
	FUDGE	31	43	0
mxml v2.12	*IntelliGen*	735	502	5
	FUDGE	31	42	0
libevent	*IntelliGen*	414	11	0
	FUDGE	113	3	0

The statistics show that *IntelliGen* covers more blocks and paths and can detect more bugs than *FUDGE* in all cases tested. This is because *IntelliGen* tends to synthesize the fuzz drivers on high-level entry functions while *FUDGE*'s synthesis criteria is limited to a specific function signature. In addition, *IntelliGen* and FUDGE cover more blocks and paths on *mxml v2.9* and *mxml v2.12* than on *libevent*. This is because **libevent** is stateful, thus requiring the driver to call a series of functions in a particular order before starting the whole project. One way to improve this is to let *IntelliGen* learn the order in which to call multiple entry functions from unit test cases.

VII. Lesson Learned

Writing a fuzz driver manually seriously hinders the efficiency of fuzz testing. Fuzzing tools have been well developed and widely deployed in industrial environments and have detected many vulnerabilities. However, the overall performance and effectiveness of fuzz testing is still below expectations, since testers need to undertake the extremely laborious task of constructing fuzz drivers manually for each project. This renders fuzz testing inaccessible to many who wish to use fuzz testing. Generating fuzz drivers automatically can greatly reduce the amount of manual labor required, especially for large projects, which usually demands a diverse portfolio of fuzz drivers to cover most areas of the program's code.

The quality of fuzz drivers will drastically impact the performance of fuzz testing. A fuzz driver should select a high-value entry function to maximize its effectiveness. Different entry functions can reach different parts of the code and will determine the direction of the fuzzing. A high-level function usually calls many low-level functions, thus calling a high-level function usually covers more branches and paths than that of calling a low-level function. However, sometimes a high-level function may contain numerous error checking code, making it difficult for the fuzzer to reach low-level code. An efficient fuzz driver should attempt to bypass these error checking code and reach the core of a project directly. The experiment results show that the choice of entry functions has a great influence on fuzzing efficiency, and we need more intelligent methods to decide the entry function.

The performance of driver synthesis can be improved with more domain knowledge.

As we have explained previously, some projects are stateful and will require the fuzz driver to call a series of functions in a specific order to reach most of the code. In addition, our argument construction algorithm may produce semantically incorrect entry function parameters. Through leveraging domain knowledge automatically extracted from test cases and other programs that use this library, *IntelliGen* can solve the aforementioned problems through learning an entire sequence of function calls and understanding which function parameters would be accepted by the library, respectively.

The criteria for identifying effective entry functions is largely undetermined. Though *IntelliGen* utilizes the *Entry Function Locator* to locate potential entry functions using metrics that represent potential memory vulnerabilities, to avoid generating sub-optimal fuzz drivers, we still manually select the entry functions that potentially allow the fuzz engine to cover a large amount of the code. A concrete selection criteria is needed to further filter out less potential entry functions and maximize the effectiveness of generated fuzz drivers. Nevertheless, our evaluation is valid regardless of this procedure, as a fully automated procedure would simply generate more fuzz drivers.

VIII. Conclusion

In this paper, we propose *IntelliGen*, an automated fuzz driver synthesis framework. It constructs valid fuzz drivers using the following steps. *IntelliGen* first calculates the vulnerability priority for each function and takes the functions with the highest priority as entry functions. Then, *IntelliGen* mutates the arguments of functions called in an entry function and synthesizes parameters for the entry function using algorithmic parameter construction to synthesize a fuzz driver. We evaluate *IntelliGen*'s effectiveness against state-of-the-art fuzz driver synthesizers *FuzzGen* and *FUDGE* on real-world projects selected from the Android Open Source Project, Google's *fuzzer-test-suite* and our industrial collaborators. Compared with *FuzzGen*, *IntelliGen* covers 2.03× more blocks and 1.36× more paths. Compared with *FUDGE*, *IntelliGen* covers 1.08× more blocks, 2.06× more paths, and detects ten more bugs. *IntelliGen* constitutes a significant improvement over the current state-of-the-art, providing vastly improved driver synthesis quality with a vast reduction in manual intervention, making fuzz testing more accessible and versatile.

References

[1] American fuzzy lop. https://lcamtuf.coredump.cx/afl/.

[2] Cluster fuzz document. https://github.com/google/oss-fuzz/blob/master/docs/clusterfuzz.md.

[3] Continuous fuzzing for opensource softwar. https://github.com/google/oss-fuzz/blob/master/docs/clusterfuzz.md.

[4] Google honggfuzz. https://github.com/google/honggfuzz.

[5] Libfuzzer. http://llvm.org/docs/LibFuzzer.html.

[6] Babić, D., Bucur, S., Chen, Y., Ivančić, F., King, T., Kusano, M., Lemieux, C., Szekeres, L., and Wang, W. Fudge: fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2019), pp. 975–985.

[7] Baresi, L., and Miraz, M. Testful: Automatic unit-test generation for java classes. In *2010 ACM/IEEE 32nd International Conference on Software Engineering* (2010), vol. 2, IEEE, pp. 281–284.

[8] Böhme, M., Pham, V.-T., and Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering 45*, 5 (2017), 489–506.

[9] Chen, P., and Chen, H. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018), pp. 711–725.

[10] Chen, P., Liu, J., and Chen, H. Matryoshka: Fuzzing deeply nested branches. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2019), CCS '19, Association for Computing Machinery, p. 499–513.

[11] Chen, Y., Jiang, Y., Ma, F., Liang, J., Wang, M., Zhou, C., Jiao, X., and Su, Z. Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *28th {USENIX} Security Symposium ({USENIX} Security 19)* (2019), pp. 1967–1983.

[12] Fu, Y., Ren, M., Ma, F., Shi, H., Yang, X., Jiang, Y., Li, H., and Shi, X. Evmfuzzer: detect evm vulnerabilities via fuzz testing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2019), pp. 1110–1114.

[13] Gao, J., Xu, Y., Jiang, Y., Liu, Z., Chang, W., Jiao, X., and Sun, J. Em-fuzz: Augmented firmware fuzzing via memory checking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 39*, 11 (2020), 3420–3432.

[14] Ispoglou, K., Austin, D., Mohan, V., and Payer, M. Fuzzgen: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20)* (Boston, MA, Aug. 2020), USENIX Association.

[15] Jaygarl, H., Lu, K.-S., and Chang, C. K. Genred: A tool for generating and reducing object-oriented test cases. In *2010 IEEE 34th Annual Computer Software and Applications Conference* (2010), IEEE, pp. 127–136.

[16] Kampmann, A., and Zeller, A. Carving parameterized unit tests. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)* (2019), IEEE, pp. 248–249.

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

[18] Liang, J., Chen, Y., Wang, M., Jiang, Y., Yang, Z., Sun, C., Jiao, X., and Sun, J. Engineering a better fuzzer with synergically integrated optimizations. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)* (2019), IEEE, pp. 82–92.

[19] Liang, J., Jiang, Y., Chen, Y., Wang, M., Zhou, C., and Sun, J. Pafl: extend fuzzing optimizations of single mode to industrial parallel mode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2018), pp. 809–814.

[20] Liang, J., Jiang, Y., Wang, M., Jiao, X., Chen, Y., Song, H., and Choo, K.-K. R. Deepfuzzer: Accelerated deep greybox fuzzing. *IEEE Transactions on Dependable and Secure Computing* (2019).

[21] Liang, J., Wang, M., Chen, Y., Jiang, Y., and Zhang, R. Fuzz testing in practice: Obstacles and solutions. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2018), IEEE, pp. 562–566.

[22] Luo, Z., Zuo, F., Jiang, Y., Gao, J., Jiao, X., and Sun, J. Polar: Function code aware fuzz testing of ics protocol. *ACM Transactions on Embedded Computing Systems (TECS) 18*, 5s (2019), 1–22.

[23] Luo, Z., Zuo, F., Shen, Y., Jiao, X., Chang, W., and Jiang, Y. Ics protocol fuzzing: coverage guided packet crack and generation. In *2020 57th ACM/IEEE Design Automation Conference (DAC)* (2020), IEEE, pp. 1–6.

[24] Lyu, C., Ji, S., Zhang, C., Li, Y., Lee, W.-H., Song, Y., and Beyah, R. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th {USENIX} Security Symposium ({USENIX} Security 19)* (2019), pp. 1949–1966.

[25] Ma, L., Artho, C., Zhang, C., Sato, H., Gmeiner, J., and Ramler, R. Grt: Program-analysis-guided random testing (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2015), IEEE, pp. 212–223.

[26] Nguyen, T. T., Nguyen, H. A., Pham, N. H., Al-Kofahi, J. M., and Nguyen, T. N. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering* (2009), pp. 383–392.

[27] Pacheco, C., Lahiri, S. K., Ernst, M. D., and Ball, T. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)* (2007), IEEE, pp. 75–84.

[28] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. Addresssanitizer: A fast address sanity checker. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)* (2012), pp. 309–318.

[29] Shi, H., Wang, R., Fu, Y., Wang, M., Shi, X., Jiao, X., Song, H., Jiang, Y., and Sun, J. Industry practice of coverage-guided enterprise linux kernel fuzzing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2019), pp. 986–995.

[30] Wang, L., Fang, L., Wang, L., Li, G., Xie, B., and Yang, F. Apiexample: An effective web search based usage example recommendation system for java apis. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)* (2011), IEEE, pp. 592–595.

[31] Wang, M., Liang, J., Chen, Y., Jiang, Y., Jiao, X., Liu, H., Zhao, X., and Sun, J. Safl: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (2018), pp. 61–64.

[32] Zhong, H., Xie, T., Zhang, L., Pei, J., and Mei, H. Mapo: Mining and recommending api usage patterns. In *European Conference on Object-Oriented Programming* (2009), Springer, pp. 318–343.

[33] Zhou, C., Wang, M., Liang, J., Liu, Z., and Jiang, Y. Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2020), IEEE, pp. 858–870.