IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received September 6, 2019, accepted September 30, 2019, date of publication October 9, 2019, date of current version October 24, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2946444

UISFuzz: An Efficient Fuzzing Method for CPU Undocumented Instruction Searching

XIXING LI, ZEHUI WU, QIANG WEI, AND HAOLAN WU
China National Digital Switching System Engineering and Technological Research Center, Zhengzhou 450002, China
Corresponding author: Qiang Wei (prof_weiqiang@163.com)

This work was supported by the National Key Research and Development Program of China under Grant 2017YFB0803202.

ABSTRACT With the rapid development of network security and the frequent appearance of CPU vulnerabilities, CPU security have gradually raised great attention and become a crucial issue in the computer field. Undocumented instructions, as one of the important threats to system security, is an important entry for CPU security research. Using fuzzing technology can automatically test the CPU instruction set and discover potential undocumented instructions, but the existing methods are of slow search speed and low accuracy. Therefore, this paper designs an efficient fuzzing method (UISFuzz) for undocumented instruction searching. This method has the following merits: (1) the instruction search speed is greatly improved by an automatic instruction format recognition, as the low efficient part of the known instruction format is skipped and therefore the instruction search space is much narrowed; (2) the false positive rate is reduced by a recheck mechanism based on the expert knowledge database to filter the wrongly found instructions; (3) the overhead of the method is decreased by optimizing the result analysis program, and the scope of the system is expanded, where more processors with lower performance are compatible. Typical CPU experimental results show that, the UISFuzz can successfully find undocumented instructions in the CPUs and simultaneously improve the time efficiency by 5 times compared with existing tools.

INDEX TERMS Undocumented instructions, fuzz, CPU, instruction analysis.

I. INTRODUCTION

With the rapid development of electronics economy (for example, e-wallets, encrypted virtual currency and mobile payments), information technology has been strongly related to the economy and business [1]. And at the same time, the risk of the computer information system attack is increasing due to the economic benefits [2]. Thus, the security of computer information system is attracting more attention in recently years [3]. As a core unit of computer architecture [4], CPU's security has an important impact on the computer information system. The CPU vulnerabilities exploded in recent years [5]–[12] have inspired us that, computer could not work safely and properly no matter how many security mechanisms are integrated into operating system once the CPU has security problems. The severe security threats and the important position of CPU urge us to pay more attention to the current immature CPU security researches.

At present, the security testing of CPUs still is a very challenging task. Until now, most of CPU tests were carried out by their manufacturer [13], [14]. However, current CPU tests are mainly concentrated on the CPU's performance [16]–[18]. And due to trade secrets, it is generally not disclosed on whether to conduct security testing or not, how to conduct security testing and what the related test data are. On the other hand, the increasing complexity and integration of the CPU raise the difficulty for researchers and individuals of non-CPU vendors to conduct security tests on the chip [14]. Therefore, a feasible CPU detection solution is urgently needed for the computer information security industry to fill the gap.

As an important interface to connect computer hardware units and software systems, the instruction set is an excellent entry to detect whether the CPU has security flaws or not [15]. And it turned out that [9], testing the entire instruction space of the CPU to find out undocumented instruction (defined in Section III) can effectively detect CPU vulnerabilities. However, the existing researches for CPU undocumented instruction search are still insufficient.

A. RELATED WORK

In the past, cybersecurity researchers generally considered software to be untrustworthy, and designed many techniques

The associate editor coordinating the review of this manuscript and approving it for publication was Jiafeng Xie.

to find its vulnerability or backdoors. However, compared to the software, the research on CPU security is limited. Although through the history of CPU development, it is obvious that vulnerabilities emerge in endlessly. Therefore, this chapter will be divided into two parts: (1) the history of CPU vulnerabilities and backdoors; (2) researches for security test of software and CPU.

1) HISTORY OF CPU VULNERABILITIES AND BACKDOORS

CPU vulnerabilities and backdoors have been appearing since the computer was put into use. By summarizing literatures of CPU vulnerabilities, the CPU vulnerabilities can be categorized into two types: (1) vulnerabilities that implemented in the CPU design architecture (Memory Sinkhole [7] in 2015, Meltdown [10], Spectre [11] and TLBleed [12] in 2018); (2) vulnerabilities that executed in the CPU instruction set (FDIV [5] in 1994, F00F [6] in 1997 and ''RosenBridge'' [8] in 2018). The former vulnerabilities are caused by logical defects in the CPU design architecture [1]. This type of vulnerabilities requires researchers to have a deep knowledge of CPU architecture and implementation, and it is difficult to automate and formalization [11]. The second type is caused by inconsistence or deficiency of the CPU manual description [8], which can be found by Undocumented Instruction Search. The instruction set is an interface provided by the CPU to the software and operating system layers, which can be directly called and operated [4]. Therefore, many mature technologies in software security can be applied to CPU instruction set analysis [1]. And this paper also adopts the Undocumented Instruction Search to find suspicious instructions of potential CPU vulnerabilities.

2) RESEARCHES FOR SECURITY TEST OF SOFTWARE AND CPU

There are many methods to discover vulnerabilities and backdoors in software, which can be divided into two categories: program static vulnerability analysis techniques and dynamic vulnerability analysis techniques [41]. The static vulnerability mining method represented by ITS4 [21] and MOPS [22] is simple and easy to understand, but it is of high false positive rate and poor applicability [41]. The dynamic vulnerability mining method represented by fuzzing is widely-deployed [24] since its introduction in the early 1990s [23], it has evolved from simple robust test to the self-feedback based on the code coverage or else information from the execution process [27], [28]. And with the development of artificial intelligence, vulnerability analysis and exploit technology, fuzzing technology has gradually become more intelligent [29], and the entire software vulnerability discovery technology is gradually mature and moving towards automation [30], [31].

Compared with the software vulnerability discovery, the research of CPU security is still lack and fragmented [1]. In 2015, Hicks *et al.* [32] summarized the past processor bugs and proposed a lightweight runtime mechanism for protecting software from security-critical processor bugs. But it is severely limited as it relies on open source processors. In 2014, Chen and Ahn [33] conducted a security analysis of the microcode of the x86 processor, and pointed out its possible attack, but did not give a solution for it. On the 2017 USENIX Security, Koppe *et al.* [34] reversed engineer the microcode of x86 processor, completed the custom microcode update, and implemented a Trojan in microcode level, but with the update of microcode, its method expired.

In the existing published researches, attention directly concentrated on undocumented instruction is limited [1]. In 2009, DUFLOT L. [35] proved that undocumented instructions pose a threat to the computer system security by embedding the code into the virtual machine to simulate the CPU undocumented backdoor, but it did not give a way to solve such problems. Sandsifter [9] can be regarded as the first automatic tool focusing on the undocumented instruction search, which can efficiently avoid invalid instruction space, but the method still has limitations on efficiency and false positive rate, which is difficult to be practical. And the sandsifter tool and its thesis were published at the Blackhat conference, which favors hacker technology, and it not attracted enough attention from the academic community. Zhu *et al.* [1] proposed the CPU Security Benchmark, and mentioned that their undocumented instruction search tool was improved compared to Sandsifter. However, there are no detail about their theoretical models, system design or experimental results.

B. OUR CONTRIBUTION

Therefore, to meet the requirements of CPU security testing, this paper designs and implements an efficient fuzzing method (UISFuzz) for CPU undocumented instruction searching. Firstly, aiming at the undesirable problem of violent searching with huge instruction space, a depth-first search algorithm based on instruction prefix matching is proposed to identify valid instruction boundaries, so that the instruction space decrease to a solvable range. Then, to blindness caused by the fact that the instructions to be tested are regarded as byte arrays in the search process, an automatic instruction format recognition are adopted, where the byte bits with low contribution to instruction behaviors are extracted and then skipped or sampled according to configuration. Finally, aiming at the false alarms in the existing detection methods, an instruction recheck mechanism based on the expert knowledge base is introduced. To sum up, this paper has the following contributions:

(1) a complete and extensible solution for undocumented instruction searching is proposed, which provides a platform for follow-up researchers to continue to expand research.

(2) a new search acceleration method based on instruction format identification is proposed, which greatly reduces the search space and improves the efficiency of the algorithm.

(3) an instruction recheck mechanism for undocumented instruction search is proposed, which improves the correctness of the algorithm.

(4) a UISFuzz prototype is implemented, which shows that it can effectively discover undocumented instructions after

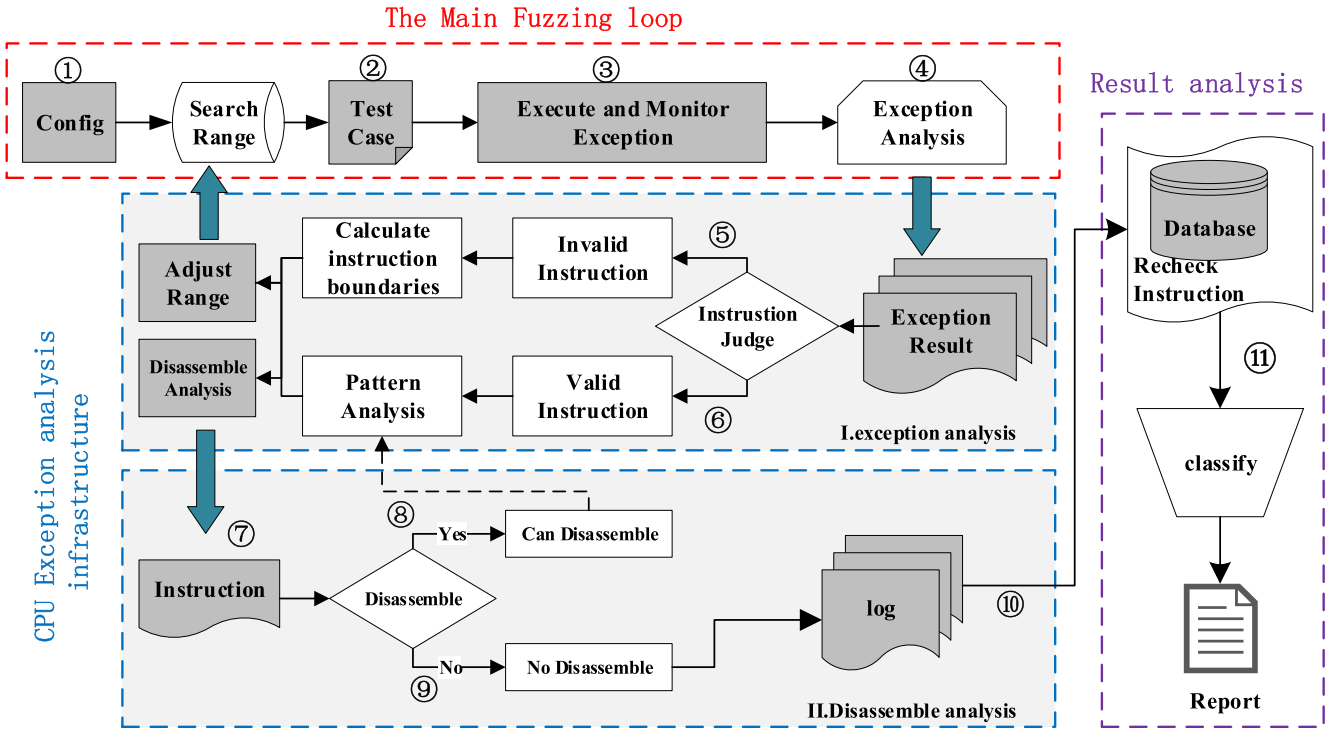

FIGURE 1. Detail steps of UISFuzz.

actual testing on various typical CPUs, and the speed was 5 times faster than the existing methods.

C. PAPER ORGANIZATION

The organization of this paper is as follows: Section II giving a macro introduction about the proposed solution and the specific steps of UISFuzz. Section III provides a detailed explanation of the specific key technologies in the UISFuzz implementation. Section IV presents the conducted experimental evaluation of UISFuzz and the performance analysis of the UISFuzz system from several different perspectives. Section V summarizes the full text and looks forward to the research that needs to be carried out later.

II. OVERVIEW

In this section, our framework of UISFuzz is presented in Part. A; and then, the running steps of UISFuzz are explained in detail in Part. B. It is worth noting that for better understanding, related definition is introduced in Part A, Section III.

A. FRAMEWORK OF UISFUZZ

In this section, by referring to fuzzing technologies in software, the framework of UISFuzz is designed and introduced. As shown in Fig. 1, in a high level, our UISFuzz model is composed of three related parts: the main fuzzing loop, the CPU exception analysis infrastructure and result analysis, whose key technologies will be explained in detail in section IV.

The main fuzzing loop, as the main process, is the overall framework for searching the instruction space, where a depth-first search algorithm based on instruction prefix matching is applied to avoid invalid instructions and narrow the search space.

Differed form the main fuzzing loop, the CPU Exception analysis infrastructure is more independent. It is mainly responsible for the exception analysis and disassembly analysis. Two jobs are done in the module: (1) compare the information of its disassembly and actual execution to determine whether it is an undocumented instruction; (2) conduct the instruction format analysis and skip the useless byte insignificant to the search.

Result analysis is responsible for filter preliminary results by recheck mechanism, classify the filtered instructions, and finally obtain a test report. It is designed to ensure the correctness.

B. RUNNING STEPS OF UISFUZZ

In this Section, the running steps of UISFuzz are described in detail. As seen in Fig.1, the workflow of UISFuzz is as follows:

•(1) **Configuration**: after the instruction search space of the target CPU is confirmed, the screen display is initialized, and then, the target instruction type, timer, log recording methods and other system performance configurations is set.

•(2) **Generation**: a machine code from the search space is selected as the candidate instruction, and then passed to the execution module.

•(3) **Execution**: after receiving the candidate instruction, the execution module loads the instruction from the predetermined initial state, and then monitor and record it whether is an exception generated or not.

•(4)**Instruction Exception Analysis**: the exception result is received by the instruction exception infrastructure module as an input to analyze whether it is valid (can be actually executed by the CPU).

•(5) **Invalid instruction condition**: if the instruction is invalid, the boundary of the valid instruction is calculated by analyzing its exception type and machine code, and then back to step (2), and guides step (2) to obtain a valid instruction.

•(6) **Valid instruction condition**: if the instruction is valid, the instruction is disassembled and analyzed to extract the operand information, and then back to step (2), and guides step (2) to speed up with the operand information.

•(7) **Disassembly Analysis**: in the disassembly analysis process, the valid instruction is used as the input of the disassembler to estimate whether it can be disassembled or not.

•(8) **Instruction Format Analysis**: for disassembled instructions, an instruction format analysis is required, and the obtained operand range is used to accelerate the instruction generation. This module is actually integrated in the disassembly analysis module in the engineering implementation.

•(9) **Judge**: by comparing the disassembled result and exception signal result of the instruction during the actual execution, the instruction is determined whether is suspicious or not and is recorded in the log.

•(10) **Recheck**: the screen display thread updates the interface display in real time according to the current exception analysis result and the disassembly analysis result of the current instruction.

•(11) **Classify**: When the search space is completely traversed, the program ends, and the detected suspicious instructions are classified and summarized as a detailed report for subsequent analysis.

III. IMPLEMENTATION

As mentioned in Section II, we implemented the prototype UISFuzz. In this section, we will describe some vital techniques in the UISFuzz implementation. Firstly, some related definitions are firstly introduced in Part. A. And then, the main fuzzing loop, the CPU exception analysis infrastructure and the recheck mechanism are respectively introduced in Part. A, Part. B, and Part. C.

A. DEFINITION

For better illustration of our method UISFuzz, some related definitions are explained in this section.

Machine Code: a string of hexadecimal byte combinations at a specific length.

Disassemble: a process that interpret machine code as a corresponding instruction as defined in the CPU manual.

Valid Instruction: Specific machine code that can be executed by the CPU. As shown in Fig.2, x86 instructions can be between 1 and 15 bytes long.

push esp
54
lock add qword ptr cs: [eax + eax*4 + 0x78563412], 0x40302010
2e67f048818480123456781020304 0

FIGURE 2. Machine code and valid instruction.

Invalid Instruction: Specific machine Code that cannot be executed in the CPU, such as 0F24, is not defined in the CPU manual and cannot be executed during actual test.

Undocumented Instruction: an instruction that can actually be parsed and executed in the CPU but is not involved in the document declared by the CPU manufacturer. The relation is shown in Equation 1.

$$\begin{aligned} \mathrm{E} &= \{x | x \text{ is executable}\} \\ \mathrm{D} &= \{x | x \text{ is documented}\} \\ \mathrm{U} &= \{x | x \in E \text{ and } x \notin \mathrm{D}\} \end{aligned} \tag{1}$$

Candidate Instruction: a specific machine code generated by the main fuzzing loop which may be a potential instruction but need to be test.

From definition, it can be clarified that an undocumented instruction has two necessary conditions: (1) executable and (2) undocumented. Thus, the problem to be solved in this paper is how to efficiently find the instruction that satisfies the two conditions in the instruction space. However, algorithms with violent enumerations can't afford this task at present, as the search space of x86 CPU is $256^{15} = 2^{120} = 1.329228\mathrm{E}36$. So, we design UISFuzz for undocumented instruction searching.

B. THE MAIN FUZZING LOOP

As mentioned above, it is impossible to complete the instruction search by pure blasting method at current computation power, so a depth-first search algorithm based on instruction prefix matching is applied to search the instruction space in the main fuzzing loop.

Due to the change of x86 instruction length, the search space is all possible for 15 bytes. After a machine code is confirmed as a valid instruction, assume that the valid instruction length is L1, the fixed first L1 bytes and any possibility of last (15-L1) bytes cannot make up any new valid instruction, so we don't need to search these bytes. Meanwhile, if a machine code is an invalid instruction, its subsequent machine code is also invalid, so we don't need to search the following bytes, too.

As the example shown in condition A of the Fig.3, {00,00} is a valid instruction, which is interpreted in the CPU as the instruction "add byte ptr [rax], al", its length is 2. So {00,00, *, *} (* means the byte can be any value) will not be valid instruction, so we can skip the subsequent 13 bytes if the

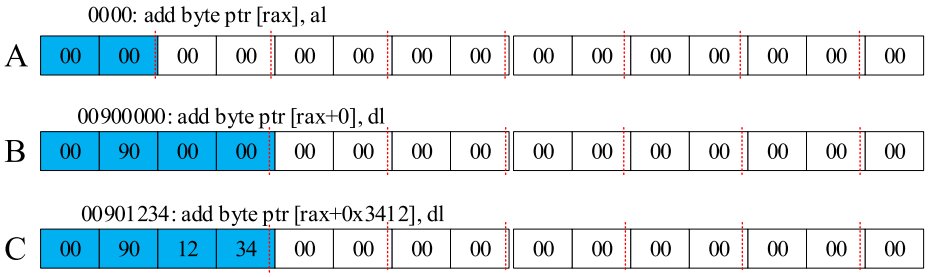

FIGURE 3. Instruction example.

{00,00} not change. Similarly, we can skip the subsequent 11 bytes in the condition C of the Fig.3.

Inspired by the above rule and based on the method from sandsifter [9], a 15-byte buffer is firstly set and assigned to 0 as the candidate instruction. After the first execution of the candidate instruction, the valid instruction part is obtained, and then the valid instruction is divided into two parts: the prefix part and the variant part. The initial prefix part length is 1, and the variant part length is 1. As shown in Fig. 4, the yellow byte is the prefix part and the green byte is the variant part. At each time, the prefix part remains unchanged, change the variant part from back to the front while the valid instruction length is observed. If the length of valid instruction has changed, extend the variant to the new instruction length, continue change the variant part from back to the front. In each change, the corresponding byte's value is increased by 1. If the byte value become "0xFF" ("0xFF" is a hexadecimal representation of 255, because we change the byte from 0 to 255 incrementally, so if one byte value finally became "0xFF", that means 256 possibilities of this byte has been exhausted.), then backtrack one byte. When the first byte become "0xFF", the search process ends. The search logic is that under the same prefix conditions, look for longer instructions first, until the conditions "0xFF" are met and then backtracking. The depth-first search algorithm is an algorithm for traversing or searching tree or graph data structures that starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking [42]. So, this search algorithm was named "a depth-first search algorithm based on instruction prefix matching".

An example is shown in Fig.4. In the status ① ,the first byte is the prefix part, the second byte is the variant part. So, from status ① to status ④, the value of second byte changes from "0x00" to "0x03". When it changes to status ⑤, the instruction length changes, so the first two bytes become the prefix part and the third byte become the variant part. And continue to change the third byte until to status ⑨, the instruction length changes again. Then, in the same way, the first three bytes make up the new prefix part while the rest fourth byte turn to the variant part. Then the last four bytes continually change until the corresponding byte's value become "0xFF" and backtrack. And in the status ⑮, all the possibilities of variant part have been searched. So, it backtracks to the third byte. And at a certain time, it finds the longest instruction at the condition I. At the end, when the first byte turns to "0xFF", the whole search ends. Therefore, through such an algorithm, invalid instructions were heavily reduced, which make searching the instruction space turn to feasible.

C. THE CPU EXCEPTION ANALYSIS INFRASTRUCTURE

The important premise of a depth-first search algorithm based on instruction prefix matching mentioned in Section B is that the instruction length can be accurately parsed for any machine code instruction. In this section, an exception analysis to get instruction lengths is firstly introduced in Part. 1, and an instruction format recognition to speed up search progress is introduced in Part. 2.

1) GET INSTRUCTION LENGTH BASED ON EXCEPTION ANALYSIS

The CPU provides an exception handling mechanism for handling various unintended conditions during instructions execution. An exception happened indicates a problem with the current instruction. For example, if the current instruction attempts to divide by 0, the CPU will throw an exception, and the CPU immediately interrupts its current work and calls a specific exception handler according to the exception's type. There are about 20 CPU exception types in the x86 architecture. Some exceptions information related to this paper like Page Fault (#PF), Invalid Opcode (#UD), General Protection Fault (#GP) and else are listed in Table 1.

TABLE 1. Some Exceptions information (Excerpt from intel architectures software developer's manual).

VECTOR	MNEMONIC	DESCRIPTION	Source
0	#DE	Divide Error	DIV and IDIV instructions
4	#OF	Overflow	INT0 instruction
6	#UD	Invalid Opcode	UD instruction or reserved opcode
13	#GP	General Protection	Any memory reference and other protection checks
14	#PF	Page Fault	Any memory reference

By monitoring the exception during the CPU execution of the instruction, we can get some information about the

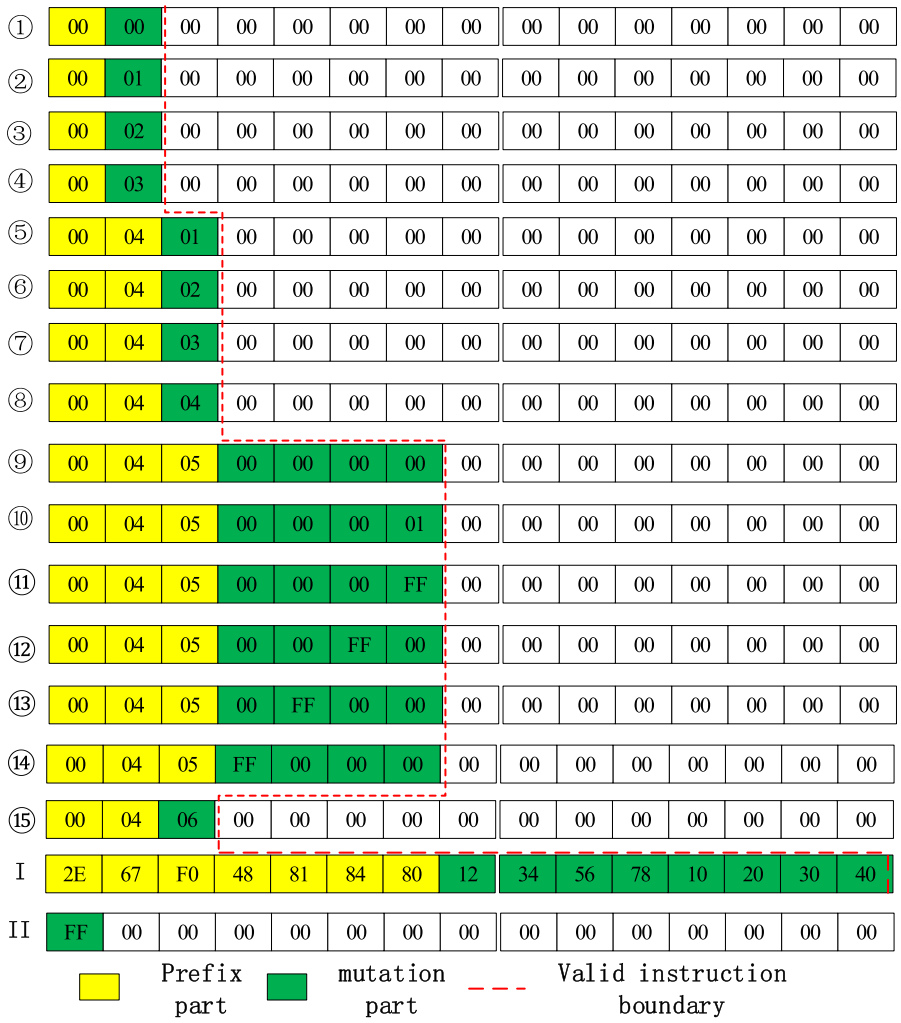

FIGURE 4. Depth-first search based on instruction prefix matching (Based on the method from sandsifter).

instruction from the side. For example, if the #UD exception is generated during the execution of candidate instruction, the candidate instruction must be an invalid instruction. If there is no exception during execution, or exception is generated but not #UD, the candidate instruction is a valid instruction. At the same time, when the CPU attempts to execute an instruction from a non-executable memory page (An operating system with support for the NX [43] bit may mark certain areas of memory as non-executable. The processor will then refuse to execute any code residing in these areas of memory), a #PF exception is thrown and the CR2 register is set to the start address of the non-executable page.

The specific method is as follows, first allocate two adjacent memory pages, the previous memory page is set as an executable page, the latter memory page is set as a non-executable memory page. Then first byte of the candidate instruction is placed on the last byte of executable page; The remaining bytes are placed in subsequent non-executable pages in order. The candidate instruction is the executed, if the CPU trigger the #PF exception, and the start address of non-executable page is recorded in the CR2 register, that means part of the candidate instruction lies in the non-executable page. Any other results mean that the whole valid instruction is fetched and executed. Therefore, we continue to move the candidate instruction one byte forward and execute it until no #PF error or #PF error is generated during execution but the CR2 register is not the start address of non-executable page. Finally, the effective length of each candidate instruction is the length of the candidate instruction remaining in the executable page.

An example is shown in the "①get instruction length" in Fig.6. Based on the above method, the candidate instruction {26, 0F, 70, 9C, 10, 20, 30, 40, 50, 60, 90, 00, 00, 00, 00} is continuously shifted and executed. When the candidate instruction is shifted to the position that {26, 0F, 70, 9C, 10, 20, 30, 40, 50, 60} is located in the executable page while {90, 00, 00, 00, 00} is located in the non-executable page, no exception is generated during the execution, therefore the

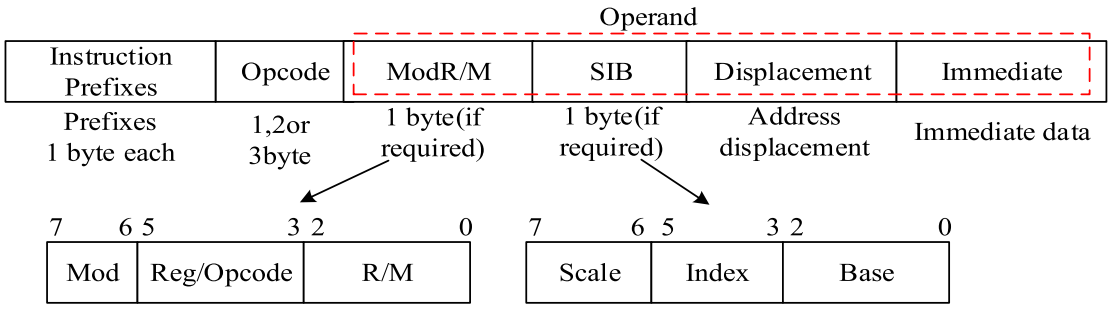

FIGURE 5. x86 instruction format (Excerpt from intel architectures software developer's manual).

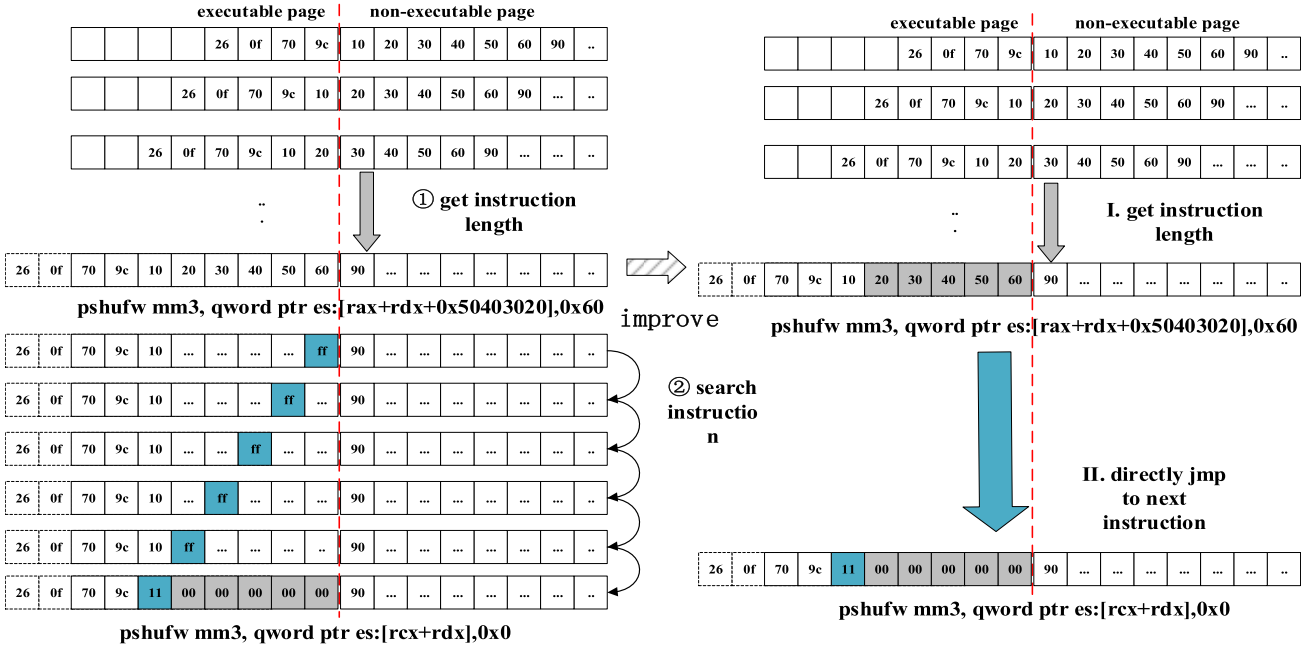

FIGURE 6. Instruction format recognition for speeding up search progress.

valid instruction length is 10 (the length of {26, 0F, 70, 9C, 10, 20, 30, 40, 50, 60} because it stay in the executable page). This method provides support for depth-first search algorithm Based on Instruction Prefix Matching in section B.

2) SPEED UP SEARCH PROGREE BASED ON INSTRUCTION FORMAT RECOGNITION

Candidate instructions are treated as a byte array in the search process until now, but the instruction have its own intrinsic format. Taking the x86 instruction set as an example, as shown in Fig.5, the machine code of a valid instruction consists of prefix, opcode, register, addressing code, offset, and immediate value. The existing method did not disassemble and analyze the valid instruction, and after obtaining the valid instruction length, it only traverse the variant part blindly. In fact, the variation of the partial bytes contributes little or almost zero to the entire search process. Such blind traversal wastes a lot of computing power and time. Therefore, this section proposes an undocumented instruction acceleration search algorithm based on instruction format recognition, which is described as follows.

Through analyzing the x86 instruction format, it is found that the opcode field only occupies up to 3 bytes, while displacement and immediate can account for up to 8 bytes. In the search progress, what we really concern is the change of opcode because the behavior of instruction is heavily depend on the opcode, as for the operand, especially immediate, has little effect on the instruction behavior compared to opcode. So, if the Immediate field in the candidate instruction can be identified and skipped during the search process, a large amount of search time can be saved.

In order to increase search speed to change to next different instruction quickly until all the instruction space is traversed. The instruction format analysis module based on disassembler is proposed to analyze the instruction format of the valid instructions. Through this module, the opcode and operand fields in the instruction format were obtained. The operand field consists of a register and an immediate value. In history, vulnerabilities like F00F happened in the

register filed. But the immediate filed has little effect on the instruction behavior while occupy high percentage of bytes. Therefore, we particularly choose to extract the corresponding machine code of immediate field in the operand, and calculate the value range of the immediate field. And then take a certain number of samples to represent it or directly skip it according to the user configuration, and concentrate the computing power on the search in the opcode and register fields which heavily affect instruction behavior. Investigating the performance improvement of the method theoretically, it is obvious to know that if the number of skip bytes in the immediate field is n, so the reduced number of search instructions is 2^{8*n}. Therefore, the instruction number of reduced in search grows exponentially with the growth of n. This mechanism is configurable and the skip range will be recorded for subsequent sole test aimed at the less probable undocumented instructions hidden in the immediate value.

An example is shown in part II of Fig.6, for the instruction {26, 0F, 70, 9C, 10, 20, 30, 40, 50, 60, 90, 00, 00, 00, 00},after the instruction length is obtained based on the exception analysis, the position of the immediate value and displacement field in the instruction operand (the last 5 bytes {20, 30, 40, 50, 60, 90}) is obtained by calling the instruction format analysis module. It is easy to calculate that the last 5 bytes can be any value from {00, 00, 00, 00, 00} to {FF, FF, FF, FF, FF}. Because the last 5 bytes only affect the displacement and immediate value of the instruction ''pshufw mm3, qword ptr es: [rax+rdx+?],? (? depended on the value of last 5 bytes.)'', the change of 5 bytes has no effect on the search progress. Assuming use skip strategy, these 5 bytes will be skipped and backtracking straightly, as shown in the part II in Fig.6.Hence, in this example, 2^{40} candidate instructions are reduced. Experiments in section V have shown that this acceleration strategy has improved search efficiency by more than five times while also find undocumented instructions correctly.

D. RECHECK MECHANISM

After above steps, a large number of suspicious instructions will be output. However, since the disassembler's own update lags behind the update of the CPU instruction manual, it is not accurate to only compare exception results and disassembly information. Therefore, a recheck mechanism based on an expert strategy database is adopted.

The databsed includes two parts: (1) updated errata, as CPU manufacturers continue to update errata to record some CPU instruction anomalies; (2) collections of instruction format analysis (including coprocessor instructions or some old obsolete instructions) organized by community on the Internet [39].The expert strategy database can filter the parts of suspicious instructions that are already documented. For example, the instruction DBE0 is the instruction in coprocessor 80287 but Sandsifter mistakenly view DBE0 as the undocumented instruction for the reason that the disassembler support the instruction in coprocessor not well. With the help of recheck mechanism, this mistake can be prevented.

After recheck mechanism, a certain count of false positives is filtered out but the number of suspicious instructions still very large (results in Section V show the number will over 10^6), which is impractical for further analysis. Thus, an automatic classification is designed. The existing method uses a recursive algorithm to analyze and classify the instructions in bytes, which is of high overload and couldn't be used on old CPUs. UISFuzz reduces the original recursive analysis to three hierarchical loops according to the instruction format: 1) combine the same instructions according to the operand field in the machine code. 2) preliminary classification of the results based on the operand fields in the machine code. 3) combine the results again according to the difference of the instruction prefix. Comparing with exist methods, the whole analysis process is automatic without manual intervention and supports the output of files and databases. It is not necessary to run this progress every time as the result is saved after once analyzed.

IV. EXPERIMENT AND DISCUSSION

To test the performance of the proposed UISFuzz, experiments are conducted on some typical CPUs with the comparison of the existing method sandsifter. The evaluation setup is firstly introduced in Part A; the found undocumented instructions are then presented in Part B and the accuracy is discussed here; the high efficiency and lower overhead of the proposed method are respectively presented in Part C and Part D; Finally, a discussion on these experiments are concluded in Part E. To provide the fairest comparison as possible, any needed setup of the compared method is equivalent.

A. EVALUATION SETUP

Through programming in Linux, the UISFuzz method for undocumented instruction search contains about 2000 lines of python codes. The injection tester for instruction search and exception analysis is implemented based on sandsifter, and about 800 lines of C code are added. In order to provide detailed information about the operands during disassembly, we also revised the well-known disassembly project ''capstone'' [37], where about 400 lines of C code have been changed.

To test the feasibility of the proposed UISFuzz, some typical CPUs including Intel, AMD, Zhao Xin and VIA are tested in the following experiments. Intel, AMD and VIA are top three x86 CPU manufacturers in terms of market share, it is reasonable to add them to research object list. As for Zhaoxin, Zhaoxin [44] is a fabless semiconductor company, created in 2013 as a joint venture between VIA Technologies and the Shanghai Municipal Government. The processors are created mainly for the Chinese market. We add Zhao Xin to research list for the reason that we want to know whether there is similarity in the processors manufactured by Zhao Xin and VIA's CPU. Besides, for the comprehensiveness of the test, the performance of the processors is range from poor to better. The configuration information of these CPUs is shown in Table 2. For better comparison, experiments of sandsifter

TABLE 2. Details of experiment platforms.

Seq	CPU	Memory	Disk
1	Intel Core i7-4770 CPU @ 3.40GHz	16G DDR3	1T 7200rpm
2	Intel Core i7-8700K CPU @ 3.70GHz	16G DDR3	1T 7200rpm
3	Zhao Xin KH-26800@2.0GHz	16G DDR3	1T 7200rpm
4	Intel Xeon CPU E3-1226 v3@ 3.30GHz	16G DDR3	1T 7200rpm
5	VIA Nano U3500 @1.00GHz	2G DDR3	400G 5400rpm
6	AMD C-50@1.00Ghz	2G DDR3	400G 5400rpm

are also conducted with the same configuration. The task of each experiment is to search all the instruction space and find undocumented instructions. And through experiments, the differences between the UISFuzz and sandsifter are compare to show the advancements of our methods. Generally, these experiments are designed to answer the following questions?

Q-1: Could the UISFuzz find suspicious instructions in these CPUs and how about the accuracy? The answer will be given in Part B.

Q-2: How about the efficiency of our search method? The answer will be given in Part C.

Q-3: How about overhead compared with the existing methods? The answer will be given in Part D.

B. UNDOCUMENTED INSTRUCTION FOUND AND HIGHER ACCURACY

In order to test the feasibility of this proposed method, the UISFuzz is applied to a series of X86 processors. Table 3 shows the found undocumented instructions of these CPUs. Here are the details and analysis.

For the Intel Core i7-4770 and Intel Xeon E3-1226 processors, the following four undocumented instructions are found:

(1) {0F, 0D}, with a total of 2416014 actual measured instructions. These instructions remained undocumented until 2016 when they were published as PRFETCHw in Intel's official manual [36]. However, the CPU was manufactured in 2013. In its manual, the reg parameter limited to 1 and other values are not described, but the instruction can be implemented in the actual CPU;

(2) 0F18, with a total of 800 actual measured instructions. Similarly, these instructions were not undocumented until 2016 when they were published as PREFETCHh in the Intel official manual, but the tested CPU was produced in 2013[40].

(3) 0F {1A-1F}, with a total of 9600 actual measured instructions. Such instructions were not documented until 2016 when they were published as Reserved-NOP in the intel official manual,

(4) 0FAE, with a total of 483 actual measured instructions. Such instructions remained undocumented until 2016 when they were published in the official manual.

(5) DF {c0-c7}, with a total number of 161 actual tested instructions. Such instructions have not been specifically described in the AMD manual until now.

As for the instruction DBE0 and DBE1 found by sandsifter, UISFuzz also found it but the instruction DBE0 and DBE1 is escaped to coprocessor instruction set. And through the intel 80287 programmer's reference manual, the instruction DBE0, DBE1 are decoded as the instruction FNENI and FNDISI. As for the instruction D6, F6 /1 and F7 /1, it has documented in the sandpile.org (which is the world's leading source for technical x86 processor information). These instructions found in these two CPU have been successfully screened out by the recheck mechanism.

The reason why these two results are same may be that both CPU were produced in 2013 and instruction set were designed in a same way. The result of Intel Core i7-8700K is also same but this CPU were produced in 2017, so these results were not undocumented instructions.

For the AMD processors, two types of instructions are found:

(1) 0F0F, with a total of 2310180 actual tested instructions. This instruction has not appeared in AMD manuals yet;

(2) DF {c0-c7}, with a total number of 161 actual tested instructions. Such instructions have not been specifically described in the AMD manual until now.

For the VIA Nano U3500 processor, besides the same 0F0D, 0F18, 0F {1A-1E} and 0FAE instructions found in Intel processor, a unique undocumented instruction namely 0FA7 {C1-C7} are also found. The total number of such instructions was 64. This kind of instructions were not defined in the VIA official manual. Interestingly, the number and type of exception instructions in ZX-KH26800 processors are very similar to those in VIA processors, and also found the 0FA7 instructions which only appear in VIA. It may be related to the cooperation between ZX-KH26800 and VIA Technologies.

Besides, the instruction DBE0 and DBE1 have been found in each CPU, but after recheck mechanism they have been screened out. Therefore, the result given by UISFuzz is more accurate than sandsifter.

C. FASTER FUZZING SPEED

In order to quantitatively evaluate the efficiency of UISFuzz, the instruction search number ''ins'' is introduced as an evaluation indicator. The instruction search number represents the number of instructions that need to test to traverse the whole instruction space. It is one of the key indexes to judge the performance of a search algorithm. Because the main frequency of CPU is fixed, the number of instructions tested in a fixed time will not fluctuate greatly. It means that, the less the number of candidate instructions to search the whole instruction space, the more efficient the search method is.

Fig.7 shows the number of search instructions changing with running time of sandsifter and UISFuzz on six experimental CPUs: i7-8700K@4.20GHz, Xeon E3-1226@ 3.30Ghz and i7-4770@3.40GHz. In the following figures,

TABLE 3. Undocumented instruction found.

i7-4770@3.40GHz	Type	0F0D	0F18	0F{1A-1F}	0FAE	DF{C0-C7}	
	Sum	2146014	800	9600	504	161	
i7-8700K@4.20GHz	Type	0F0D*	0F18*	0F{1A-1F}*	0FAE*	DF{C0-C7}	
	Sum	2146014	800	9600	504	161	
Xeon-E3-1226@3.30GHz	Type	0F0D	0F18	0F{1A-1F}	0FAE	DF{C0-C7}	
	Sum	2146014	800	9600	504	161	
AMD C-50@1.00GHz	Type	0F0F			DF{C0-C7}		
	Sum	2310180			161		
VIA Nano U3500@1.00GHz	Type	0F0D	0F18	0F{1A-1F}	0FAE	DF{C0-C7}	0FA7{C1-C7}
	Sum	2146014	800	9600	504	161	64
ZhaoXin KH26800@2.00GHz	Type	0F0D	0F18	0F{1A-1F}	0FAE	DF{C0-C7}	0FA7{C1-C7}
	Sum	2146014	800	9600	504	161	64

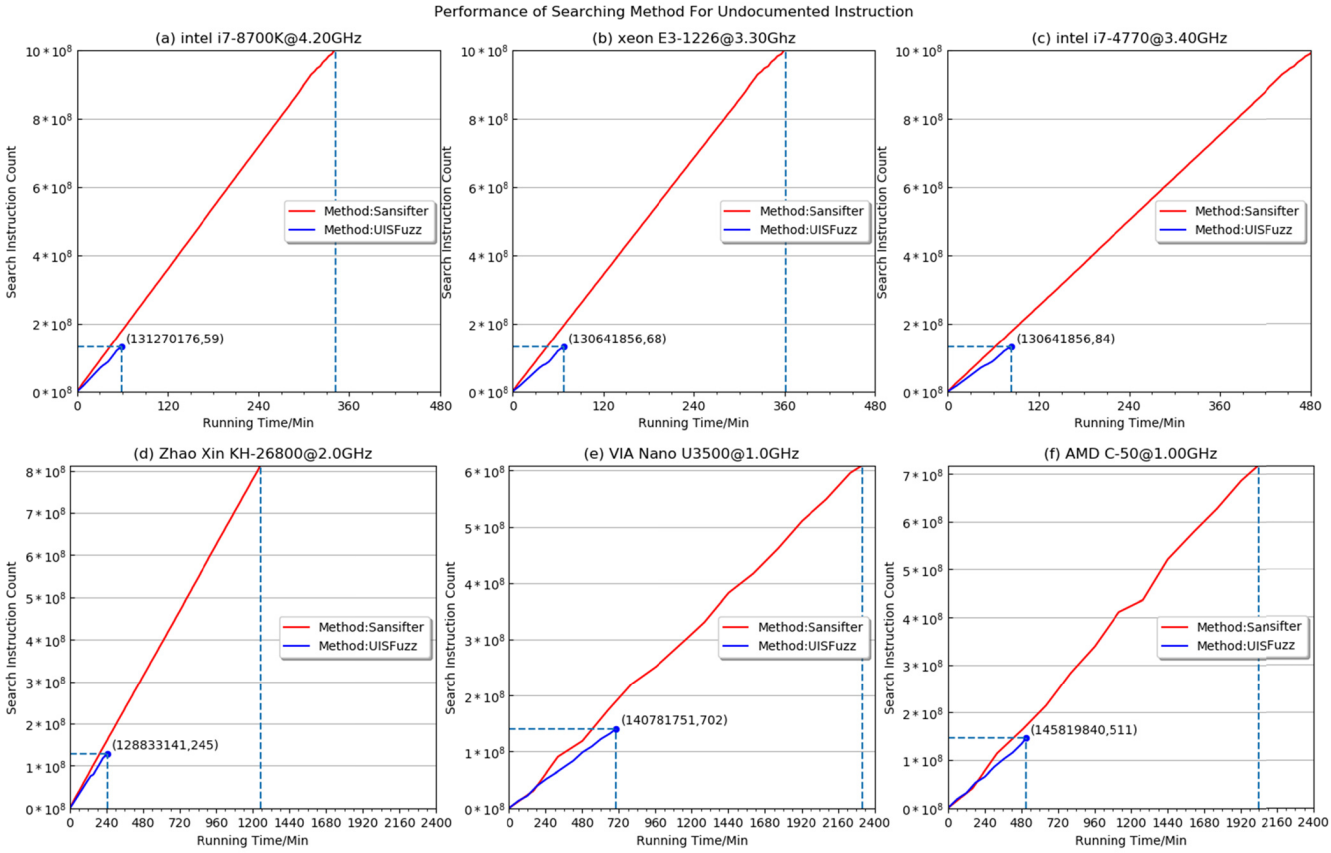

FIGURE 7. Search count and time cost of UISFuzz and sandsifter (1).

the red lines represent the results of the existing method sandsifter, while the blue lines represent the proposed UISFuzz. Especially, the blue dots represent the time when the UISFuzz completed the search of the whole instruction set space. As it obviously shows, the blue lines in the six subfigures are much shorter than the red lines, which indicates that the UISFuzz finished the instruction set space search more efficient than sandsifter. It is found that, for the same entire instruction space, our UISFuzz only needs to search for about 10^8 instructions, while the sandsifter needs to search for 10^9 instructions.

It is worth noting that the slope of the two curves in these subfigures varies little during the search process. The reason is that the CPU's main frequency is fixed, and the number of

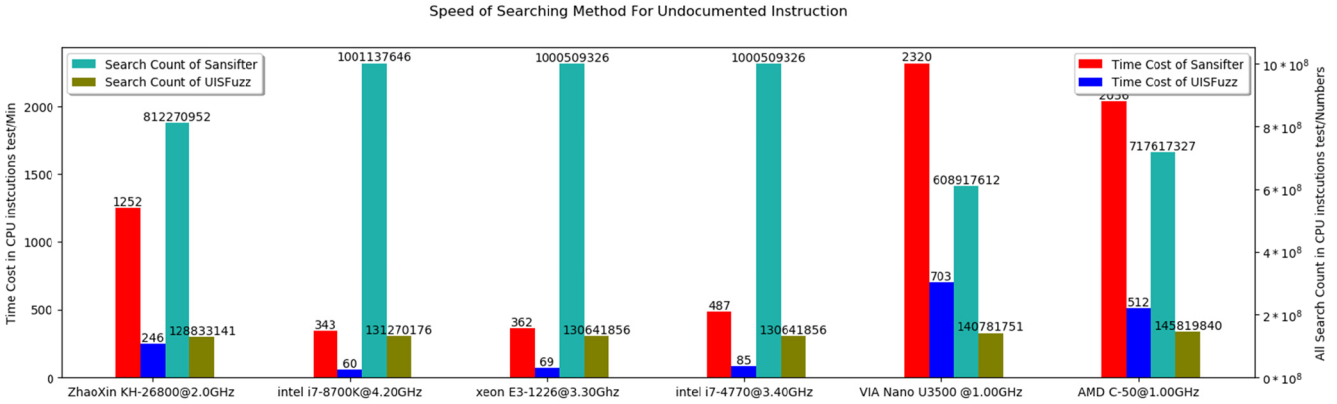

FIGURE 8. Search count and time cost of UISFuzz and sandsifter (2).

instructions tested per unit time is also fixed. On the other hand, compared to sandsifter, duo to our improved algorithm needs to do more operations for each instruction, the number of instructions that can be tested per unit time is less than sandsifter. So, the slope of UISFuzz curves is smaller than sandsifter. However, because the total number of instructions needed to search has been reduced by nearly nine times, UISFuzz can still complete the search of instructions quickly in advance.

Fig. 8 shows the total running time and instruction search number ins of existing method sandsifter and our UISFuzz on the above six CPUs. For better comparison, the runtime of UISFuzz is shown in blue while sandsifter is in red, and the instruction search number of UISFuzz is in brown, while sandsifter is in green. As Fig.8 shows, in CPU-1 (intel i7-8700k@4.20GHz), for the running time, sandsifter needs 343 minutes to complete the entire search of instructions, while our UISFuzz only needs 60 minutes, which is 5.72 times faster than sandsifter; for the instruction search number, sandsifter needs to search 1,001,137,646 instructions to complete the search of all instruction space, while UISFuzz only needs 131,270,176 searches, which reduced the instruction space by 7.62 times. The situation of the CPU-2 (intel Xeon E3-1226@3.30GHz) and CPU-3 (intel i7-4770@3.40 GHz) is similar, where the efficiency of our UISFuzz is respectively increased to 5.25 times and 5.73 times than sandsifter method, and the instruction search number of our UISFuzz is both reduced by 7.62 times than sandsifter.

In general, it can be concluded through Fig.8 that UISFuzz is at least 5.57 times faster than sandsifter in command search on the above six experimental CPUs. From the result of experiment, the search space is reduced from 10^9 to 10^8 and the search speed is 5.57 times faster than before. As for the test targeted at some software, the speed generally can be promoted by optimization of hardware. While in undocumented instruction searching, the experimental subject is CPU itself, so the promotion of speed can only rely on the promotion of the hardware. Therefore, the improvement of speed is indispensable.

D. LOWER OVERHEAD

To quantitatively evaluate the running memory of the proposed UISFuzz, the ''Memory Profiler'' [38] (Memory Profiler is a python module for monitoring memory consumption of process and is widely used in performance evaluation [38].) is applied to monitor the running time and memory consumption of sandsifter and the proposed UISFuzz on intel i7-8700k@4.20GHz and ZhaoXin KH26800@2.00GH respectively. The results are shown in Fig. 9.

Fig. 9(a) is the test result on intel i7-8700k@4.20GHz, and Fig. 9(b) is the test result on Zhao Xin KH-26800. For each subgraph, the left one is the results of sandsifter, while the UISFuzz is on the right. Fig. 9 shows that the maximum memory consumption of sandsifter is about 24,000 MB, 23.43 GB, while that of UISFuzz is only 1300 MB (about 1.26GB), which is reduced by 18.46 times compared to sandsifter. The overhead of other experiment platforms is basically similar, so we only show these two results. From Fig.9, The left curve shows that the memory cost grows like a quadratic function, While the right curve shows that memory grows in three stages corresponding to the method proposed in part C of Section IV. The load of the analyzer is very low, which is 18 times lower than the memory usage of the existing methods, and the running speed is 4 times faster than the existing methods. With the significant decrease of the overhead, the scope of application is expanded for the reason that the memory limit of some old CPUs is very small.

E. DISCUSSION

The experimental results show that the efficiency and accuracy of the proposed UISFuzz are both higher than the existing methods, while its load is also greatly reduced. Specific analyses are as follows: (1) the number of search instruction is decreased and the search speed is improved, which shows the advance of our second point in Part B, Section IV. By analyzing the instruction's format, its operand part with less relation to the operation is extracted, and the search of this part is reduced during generation, which can effectively reduce the meaningless search and make the efficiency

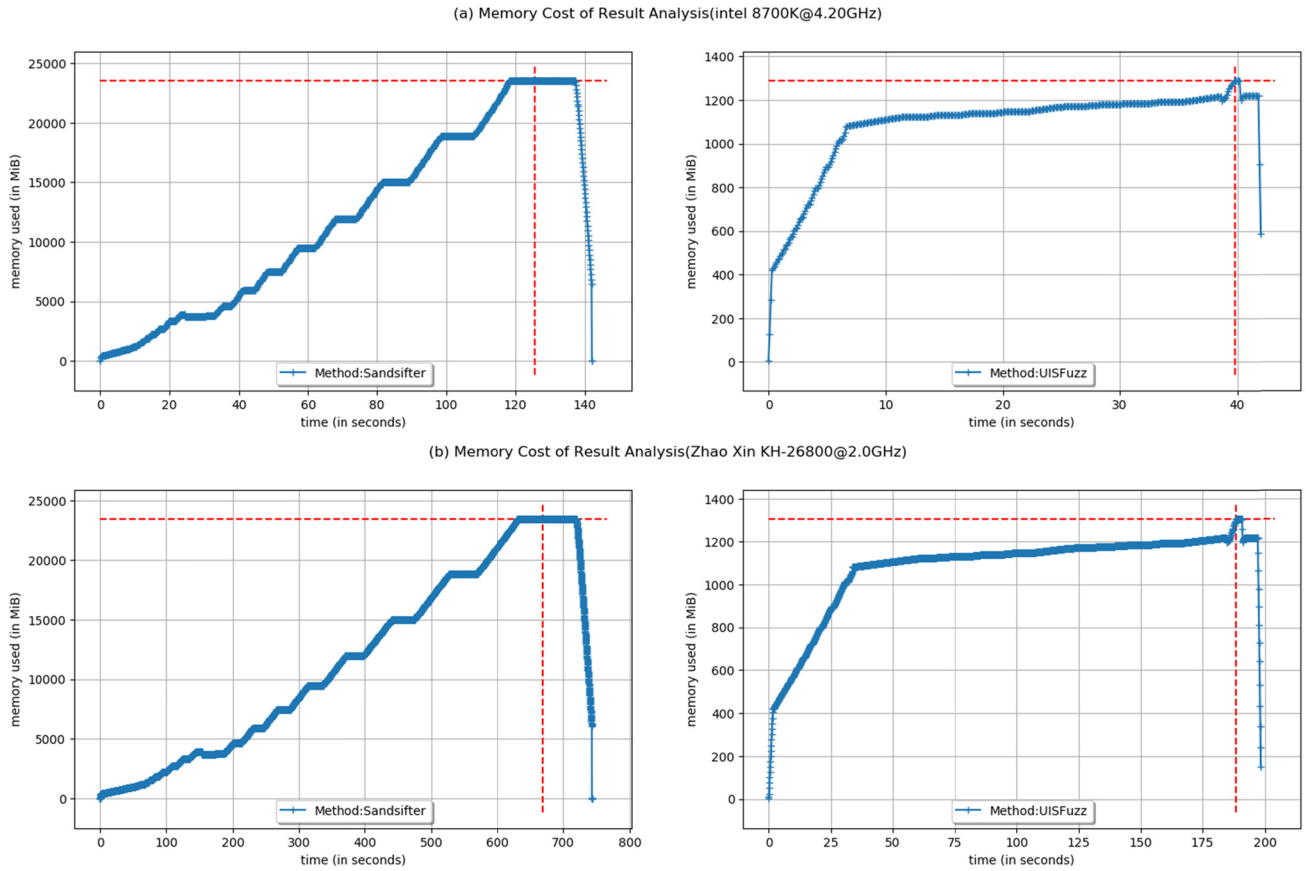

FIGURE 9. Experiment result on work overhead.

higher. (2) the accuracy of the proposed method UISFuzz is improved, which shows the validity of the recheck mechanism in Part C, Section IV. The recheck mechanism filters incorrect results by an expert knowledge database formed by regularly summarizing the published CPU errata, which can effectively ensure the accuracy of instruction search. (3) the program's overhead is much decreased, which indicates that the conversion from recursive algorithm to multiple loops in instruction structure analysis is correct and effective. In addition, the framework of UISFuzz we designed is clear, and has good guiding significance for compatibility with other platforms. However, due to time and experimental equipment, this tool now only support CPUs in the x86 architecture. More adaptation work for ARM and MIPS architectures is the follow-up research content.

V. CONCLUSION

In this paper, we design and implement an efficient fuzzing method UISFuzz for undocumented instruction search. The proposed method has several contributions: (1) the instruction format recognition is added in the instruction exception analysis process and the generation of subsequent instructions is optimized, where the variation of meaningless instructions is reduced. (2) a recheck mechanism based on the expert knowledge database was applied, which effectively reduced the false positive rate. (3) multiple loops are used to replace the recursive algorithm during instruction result analysis, reducing the memory load. Experiments show that the search efficiency has been improved 5.57 times, and the memory load has been reduced 18 times. However, there are also some shortcomings, UISFuzz currently only support CPUs in the x86 architecture, more adaptation work for ARM and MIPS architectures is need to be done. Besides, search undocumented instruction is just the first step. To judge the harmfulness of undocumented instruction, semantic analysis of undocumented instructions will be studied in the following, which will be the focus of our next work. We hope that this article can arouse more researchers' interest in CPU undocumented instructions and contribute more strength to promote CPU security.

ACKNOWLEDGMENT

The authors would like to express their gratitude to the editors and the reviewers for their constructive and helpful comments for the substantial improvement of this paper.

REFERENCES

[1] J. Zhu, W. Song, Z. Zhu, J. Ying, B. Li, B. Tu, G. Shi, R. Hou, and D. Meng, "CPU security benchmark," in *Proc. 1st Workshop Secur.-Oriented Des. Comput. Archit. Process.*, Oct. 2018, pp. 8–14.

[2] P. Brangetto and M. K.-S. Aubyn, "Economic aspects of national cyber security strategies," Project Rep. Annex 1.9-16, 2015, p. 86.
[3] B. Li, Q. Zhou, X. Si, and J. Fu, "Mimic encryption system for network security," *IEEE Access*, vol. 6, pp. 50468–50487, 2018.
[4] R. E. Bryant, D. R. O'Hallaron, S. Manasa, and M. P. Tahiliani, *Computer Systems: A Programmer's Perspective*, vol. 281. Upper Saddle River, NJ, USA: Prentice-Hall, 2003.
[5] D. Price, "Pentium FDIV flaw-lessons learned," *IEEE Micro*, vol. 15, no. 2, pp. 86–88, Apr. 1995.
[6] Pentium Processor Specifcation Update. (1999). *Invalid Operation With Locked CMPXCHG8B Instruction*. [Online]. Available: http://www.cpu-zone.com/Pentium/Pentium%20processor%20specifcation.pdf
[7] C. Domas, *The Memory Sinkhole*, New York, NY, USA: BlackHat, 2015.
[8] C. Domas, *Hardware Backdoors in X86 CPUs*. New York, NY, USA: BlackHat, 2018, pp. 1–14.
[9] C. Domas, *Breaking the X86 ISA*. New York, NY, USA BlackHat, 2017.
[10] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," Jan. 2018, *arXiv:1801.01207*. [Online]. Available: https://arxiv.org/abs/1801.01207
[11] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," Jan. 2018, *arXiv:1801.01203*. [Online]. Available: https://arxiv.org/abs/1801.01203
[12] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *Proc. 27th USENIX Symp. USENIX Secur.*, 2018, pp. 955–972.
[13] L. Fournier, Y. Arbetman, and M. Levinger, "Functional verification methodology for microprocessors using the Genesys test-program generator. Application to the ×86 microprocessors family," in *Proc. Design, Automat. Test Eur. Conf. Exhib.*, Mar. 1999, pp. 434–441.
[14] B. Bentley, "Validating the intel pentium 4 microprocessor," in *Proc. 38th Annu. Design Autom. Conf.*, 2001, pp. 244–248.
[15] S. Shamshiri, H. Esmaeilzadeh, and Z. Navabi, "Test instruction set (TIS) for high level self-testing of CPU cores," in *Proc. 13th Asian Test Symp.*, Nov. 2004, pp. 158–163.
[16] Z. Hadjilambrou, S. Das, P. N. Whatmough, D. Bull, and Y. Sazeides, "GeST: An automatic framework for generating CPU stress-tests," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2019, pp. 1–10.
[17] J. L. Henning, "SPEC CPU2000: Measuring CPU performance in the new millennium," *Computer*, vol. 33, no. 7, pp. 28–35, Jul. 2000.
[18] D. G. Feitelson, *Workload Modeling for Computer Systems Performance Evaluation*. Cambridge, U.K.: Cambridge Univ. Press, 2015.
[19] L. Chen and S. Dey, "Software-based self-testing methodology for processor cores," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 3, pp. 369–380, Mar. 2001.
[20] C. Preschern, N. Kajtazovic, A. Höller, C. Steger, and C. Kreiner, "Verifying generic IEC 61508 CPU self-tests with fault injection," in *Proc. 8th IEEE Design Test Symp.*, Dec. 2013, pp. 1–2.
[21] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw, "ITS4: A static vulnerability scanner for C and C++ code," in *Proc. 16th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, Dec. 2000, pp. 257–267.
[22] H. Chen, D. Dean, and D. A. Wagner, "Model checking one million lines of C code," in *Proc. NDSS*, vol. 4, Feb. 2004, pp. 171–185.
[23] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.
[24] V. J. M. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," Dec. 2018, *arXiv:1812.00140*. [Online]. Available: https://arxiv.org/abs/1812.00140
[25] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proc. IEEE Symp. Secur. Privacy*, May 2012, pp. 380–394.
[26] C. Cadar, D. Dunbar, D. R. Engler, and others, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Symp. Oper. Syst. Design Implement.*, vol. 8, Dec. 2008, pp. 209–224.
[27] Lcamtuf. *American Fuzzy Lop*. Accessed: May 1, 2019. [Online]. Available: http://lcamtuf.coredump.cx/a?/
[28] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path sensitive fuzzing," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 679–696.
[29] H. Xue, S. Sun, G. Venkataramani, and T. Lan, "Machine learning-based analysis of program binaries: A comprehensive study," *IEEE Access*, vol. 7, pp. 65889–65912, 2019.
[30] Z. Jiang, C. Feng, and C. Tang, "An exploitability analysis technique for binary vulnerability based on automatic exception suppression," *Secur. Commun. Netw.*, vol. 2018, May 2018, Art. no. 4610320.
[31] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou, "Revery: From Proof-of-Concept to Exploitable," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 1914–1927.
[32] M. Hicks, C. Sturton, S. T. King, and J. M. Smith, "Specs: A lightweight runtime mechanism for protecting software from security-critical processor bugs," *ACM SIGARCH Comput. Archit. News. ACM*, vol. 43, no. 1, pp. 517–529, Mar. 2015.
[33] D. D. Chen and G.-J. Ahn, "Security analysis of ×86 processor microcode microarchitecture," *Arizona State Univ.*, 2014, pp. 1–18.
[34] P. Koppe, "Reverse engineering ×86 processor microcode," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 1163–1180.
[35] L. Duflot, "CPU bugs, CPU backdoors and consequences on security," *J. Comput. Virol.*, vol. 5, no. 2, pp. 91–104, May 2009.
[36] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 3, System Programming Guide*. Accessed: Jan. 1, 2018. [Online]. Available: https://software.intel.com/sites/default/files/managed/a4/60/325384-sdm-vol-3abcd.pdf
[37] Capstone. *The Ultimate Disassembler*. Accessed: Jan. 10, 2019. [Online]. Available: http://www.capstone-engine.org/
[38] Fpedregosa. *Memory-profiler*. Accessed: Dec. 14, 2018. [Online]. Available: https://pypi.org/project/memory-profiler/
[39] MazeGen. *X86 Opcode and Instruction Reference*. Accessed: Feb. 12, 2017. [Online]. Available: http://ref.x86asm.net/.
[40] *Norm Anheier CPU-World*. Accessed: Aug. 1, 2018. [Online]. Available: http://www.cpu-world.com/
[41] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," *ACM Comput. Surv.*, vol. 50, no. 4, Nov. 2017, Art. no. 56.
[42] Wikipedia. *Depth-First Search*. Accessed: Aug. 2, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Depth-first_search
[43] Wikipedia. *NX Bit*. Accessed: Aug. 2, 2019. [Online]. Available: https://en.wikipedia.org/wiki/NX_bit
[44] Wikipedia. *Zhaoxin*. Accessed: Jun. 1, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Zhaoxin