Received May 10, 2022, accepted May 29, 2022, date of publication June 8, 2022, date of current version June 15, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3181283

Efficient Generation of Program Execution Hash

EUNYEONG AHN[1], SUNJIN KIM[2], SAEROM PARK[1], JONG-UK HOU[3], (Member, IEEE), AND DAEHEE JANG[1]

[1]Convergence Security Engineering, Sungshin Women's University, Seoul 02844, South Korea
[2]Future Convergence Technology Engineering, Sungshin Women's University, Seoul 02844, South Korea
[3]School of Software, Hallym University, Chuncheon 24252, South Korea

Corresponding author: Daehee Jang (djang@sungshin.ac.kr)

This work was supported by the National Research Foundation of Korea (NRF) funded by the Korean Government (MSIT) under Grant 2021R1F1A1049957.

ABSTRACT Distributed computing systems often require verifiable computing techniques in case their node is untrusted. To verify a node's computation result, proof-of-work (PoW) is often utilized as a basis of verifiable computing method; however, this mechanism is only valid for computations producing results based on specific algorithm (e.g., AES decryption). To date, there is no efficient PoW mechanism applicable to arbitrary algorithm or a computation that does not produce any tangible output (e.g., void function). This paper proposes *execution hash* to serve as a proof for a program's idempotent computation result without relying on its algorithm. Two versions of execution hash generation methods were designed and implemented and the efficacy was evaluated in terms of performance and reliability. Implementation was based on LLVM/Clang 6.0 and evaluation was based on open-source software, including GNU binutils/coreutils and Google's OSSFuzz projects.

INDEX TERMS Binary, execution hash, proof of work, program trace, verifiable computing.

I. INTRODUCTION

With the recent implementation of cloud computing in various fields, outsourcing specific tasks to third parties have become prevalent, which has bolstered the importance of verifiable computing [1]–[4]. Verifiable computing generally refers to the process where one client outsourcing tasks, such as computation, to multiple untrusted entities. Each entity participating in the computation must verify its conduct by returning the results with proof that it executed the work correctly. However, issues that may arise include dishonest workers, not actually performing the computations, and returning plausible results [1]. Therefore, researchers have conducted various studies on enabling clients to verify, with little effort, if the work was correctly performed [5].

However, previous cases of verifiable computing primarily outsourced computations that have a clear input-output relationship, such as mathematical or cryptographic operations [6], [7]. This study proposes execution hash as a methodology that enables the client to verify the execution flow of all programs intuitively, including computations where a result does not exist (e.g., void function), or where the input-output relationship is unclear. Specifically, when offloading certain tasks to untrusted clients, the execution hash value can be used as a proof whether the computation was actually executed as intended.

The execution hash is a hash value generated from tracing the program's runtime execution flow; if the execution flow changes, then the hash value also changes.[1] There are much information related to a program execution. The execution hash can be derived from such information and one way to generate such hash is observing changes in execution flow at basic-block granularity; and by investigating changes in the frequency of calls to edges between basic blocks.

In the first method, the binary is instrumented using LLVM [8] and a hash value is generated when the binary is terminated. This hash value is result of the call records of all basic blocks executed from the beginning of the program to its termination; if the execution of even one basic block is omitted or added, or the order changes, then the hash value changes. In the second method, the binary also generates hash value but in a different way compared to the first method: a fixed size array[2] of the edges connecting the basic blocks is created, then the frequency of calls using

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

[1]The hash as we guarantee is not a hash of the code's contents but a hash of runtime execution history.

[2]In LLVM, this is referred as inline counter array.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

these edges are stored in the array, and the hash value of the array is calculated. Therefore, although the hash value will differ if the execution of even one basic block changes, the execution order of the basic blocks does not have to be considered. Thus, even if a program is identically executed at high-level semantics, these methodologies can be used to track changes in internal memory values or differences in the detailed execution flow.

This study proposes EXHGen(Execution Hash generator): a tool for efficient generation of program execution hash that can observe the program's execution flow and investigate subtle changes in single-threaded executions. As the execution hash is a trustworthy logical indicator, it can be applied to verifiable computing to save time and cost, and resolve the problem of dishonest participants. The execution hash method can easily observe the program execution flow and is applicable to numerous situations other than verifiable computing, such as deterministic execution and detailed debugging.

This study conducted an additional test that applied the execution hash to deterministic execution. Traditionally, deterministic execution is a field that generally investigates methods to effectively debug multi-threaded applications (e.g., Dthreads [9]). However, thread scheduling is not necessarily the only factor that causes a program to execute non-deterministically. For example, even single-threaded programs can have different execution flows with the same inputs and conditions depending on the heap memory status, results of Libc functions, state of the network or file system, external interrupts, and system call-handling results. As a more specific example, if the heap memory allocation function (e.g., malloc) fails due to insufficient memory resources, the value 0 is returned, for which, an exception handling code can be considered. This is a correctness issue in the field of program testing. Although there may be no difference in the result at a level where the difference in detailed flow is very high, if the detailed computation flow is important (e.g., fuzzing [10]–[13]), then the existence of a difference becomes significant. This study applied the execution hash to investigate how frequently non-deterministic executions appear in a single thread for 108 programs included in the binutils [14] and coreutils [15] packages as well as 32 programs of OSS-Fuzz. A hooking library that can deterministically change for cases with non-deterministic execution was applied and attributed to the API function and system call execution result. Through these tests, the two methodologies that implement execution hash using existing methodologies were compared; the distinctiveness and efficiency of execution hash were demonstrated.

II. BACK GROUND & THREAT MODEL

A. BASIC BLOCK

Block is a bundle of assembly words that run at once when executing a program. Among them, basic block refers to a linear code sequence that branches only at the beginning and end. The branch at the beginning is the entry point of the block that comes in to execute the basic block at another block, and the branch at the end is the part that goes out to execute the code at another block after the execution of the basic block. Due to this limited form, branch instructions cannot exist in the middle of the code constituting the basic block, and all instructions are always executed only once in order. For this reason, tracking the execution flow of basic blocks makes it very easy to analyze the flow of instructions during program execution; it also helps to track records of certain exceptional situations, such as when errors occur. EXHGen_version1, developed in this study, uses a method of storing records of executed basic blocks in memory by inserting additional codes into each basic block and tracking the entire basic block execution record. Since the execution records of basic blocks can eventually be considered the execution flow of the program, a comparison can be made between these records and non-deterministic cases can be observed.

B. COVERAGE MAP

Coverage refers to the measurement of the execution flow of the program. It is mainly used to check how much program code has been executed for a program. When executing a program, the execution record of a basic block is generally expressed as code coverage, a concept used in gcov [16] in general software engineering and AFL coverage [17] in fuzzing. EXHGen_version2, developed in this study, measures the execution flow of the program based on this coverage map. There can be various types of coverage maps, and in the case of EXHGen_version2, the frequency of calls to edges between basic blocks is measured through the LLVM's inline counter array.

C. EINTR

EINTR [18] is an interrupt signal that occurs when a system call is interrupted. Interrupt can be divided into hardware interrupts, generated by hardware, and software interrupts, caused by exceptions that occurred while the system call was running. EINTR is an error signal that occurs in a software interrupt related to a system call. It indicates that the process was interrupted by a specific signal before the function could complete the normal operation in the system call process. In this study, a non-deterministic case was observed in which the basic block execution record of a single thread changed due to exceptions to function call results (EINTR signal generation) during system call and external library functions; additional experiments were conducted to solve this with hooking.

D. DBI

In computer programming, instrumentation refers to functions for diagnosing errors or monitoring the performance of programs, such as data logging, debugging, and performance tracking. Alternatively, it can also refer to the act of inserting an analysis code into the binary. Analysis code refers to a set of code inserted by the user to observe a program's behavior without changing its original semantic. Dynamic

binary instrumentation (DBI) is a type of instrumentation that happens at run time instead of compilation time. Intel PIN (free software) is one of the standard framework that supports DBI implementation [19], [20]. When executing a binary using a PIN, the codes are translated into intermediate representation and then transformed with instrumentation. That is, the binary is not directly loaded and executed, but is executed on top of PIN engine. DBI implementation using Intel PIN is convenient as PIN provides rich APIs. However, DBI takes huge transformation overhead, thus, significantly slowing down target binary execution.

E. LIBFUZZER

Libfuzzer is an in-process, coverage-guided, and evolutionary fuzzer developed by Google as part of the LLVM project [21]. It uses evolutionary algorithms to expand coverage efficiently and in-process methods to improve fuzzing's ability. Evolutionary fuzzer uses genetic algorithms to generate random mutations from sample corpus, and mutations generated every round are used as new corpus sets. The input that increases code coverage in the generated corpus set is stored and used as a sample corpus later. Therefore, in general, there is a higher probability that evolutionary fuzzing may be more successful than when randomly generated input is used. Libfuzzer refers to performing fuzzing in units of functions of the library, not executing the entire program independently for fuzzing. Additional work is required because part of the program subject to fuzzing needs to be wrapped, but the overall performance has nevertheless improved dramatically, including speed. In this study, libfuzzer was used; however, it was modified and added to a part of the compiler to create version2, an efficient tool to measure the execution flow of the program.

F. PoW

RoT (Root of Trust) [22] is always reliable software or hardware, where all the security work of a computing system depends. Ultimately, it is TC (Trusted Computing) that is intended to achieve, and there are representative platform modules or execution environments such as TPM and TEE. A trusted execution environment (TEE) and trusted platform module (TPM) can provide hardware-based functions related to security to prove program integrity, thus enabling verifiable computing. TEE is a secure execution environment provided by the security area within the main processor. Since it executes in parallel with the operating system in an isolated environment, the confidentiality and integrity of code and data loaded into the TEE are protected. Hardware techniques that can be used to implement TEE include ARM's TrustZone [23] and Intel's SGX [24]; however, this study utilized proof-of-work (PoW) as the method to implement verifiable computing. In contrast to TEE and TPM, which implement hardware-based security, PoW [25], [26] is a consensus algorithm in which nodes endlessly repeat the process of finding a hash below a target value and prove that they participated in this work. PoW aims to prove that a node performed a certain task or computation. Additionally, it also deters data manipulation, and ensures that distributed computing can be securely implemented.

G. THREAT MODEL

The threat model in our paper is participating clients disrupting the distributed (outsourced) computation; intentionally or un-intentionally. These threats are caused by a program's non-deterministic behavior and thus can be mitigated by enforcing the determinism. There are a number of previous researches considering non-determinism as a threat [27], [28]. In this paper we leverage hash-based Proof-of-Work (PoW) concept to achieve trustworthy verification of computation, thus guaranteeing the honesty of distributed computing workers.

III. DESIGN

A. EXECUTION HASH

In verifiable computing, each entity executes a program and then verifies whether the program was correctly executed mainly based on the execution result. However, there may be programs whose output execution result cannot be checked. Even under normal circumstances, users who execute programs are forced to rely solely on the output for information about the program's execution, which makes it challenging for them to obtain information about programs that are difficult to check using the output data. Even for programs whose execution result provides an output, the user cannot obtain information about the program's execution flow for a specific input without a debugging process. This means that even if an identical result is the output when the same input is passed to the program, the user cannot determine whether the same functions were called or the same flow was executed without additional work on their part. If a distributed computing system must verify aforementioned computations, the verification becomes challenging. To address such challenges, this study proposes an execution hash. An execution hash intuitively provides information on the program's execution flow without additional debugging work. The most accurate methodology for generating execution hashes is to generate a hash by tracking the full execution flow of the program. However, given the excessively high cost, tracking and hashing all information is highly inefficient. As a solution, this study proposes generating the hash by extracting only the most essential information about the program execution.

There are two ways to generate the hash: the first is a fine-grained method that traces the entire calling order of the basic blocks, and the second is a slightly more coarse-grained method that involves hashing the code coverage information. To describe the execution hash overall, after inserting specific code into the program, information indicating the execution flow is collected and the hash is calculated based on the coverage for the collected program execution flow. Any program using these methods can easily generate an execution hash, and the hash value can be used to check whether the

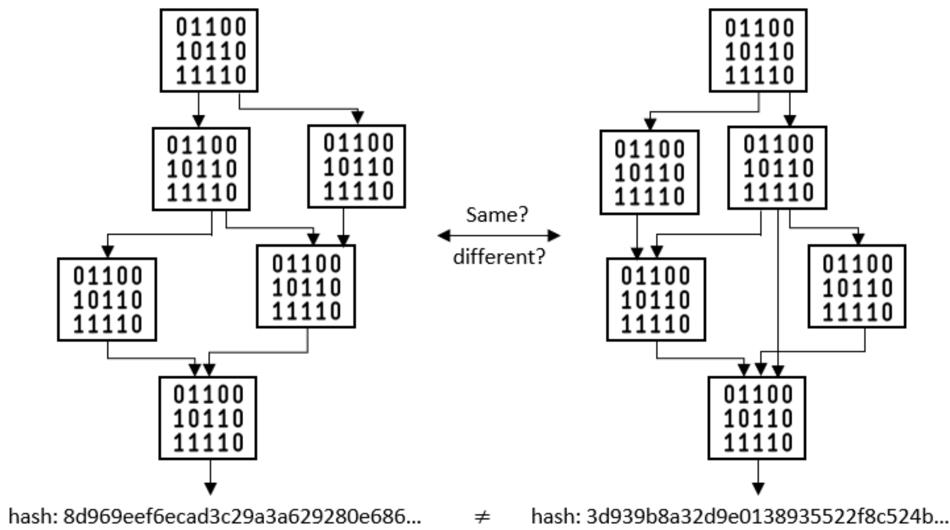

FIGURE 1. Hashing program's execution flow effectively catches subtle difference in program's behavior.

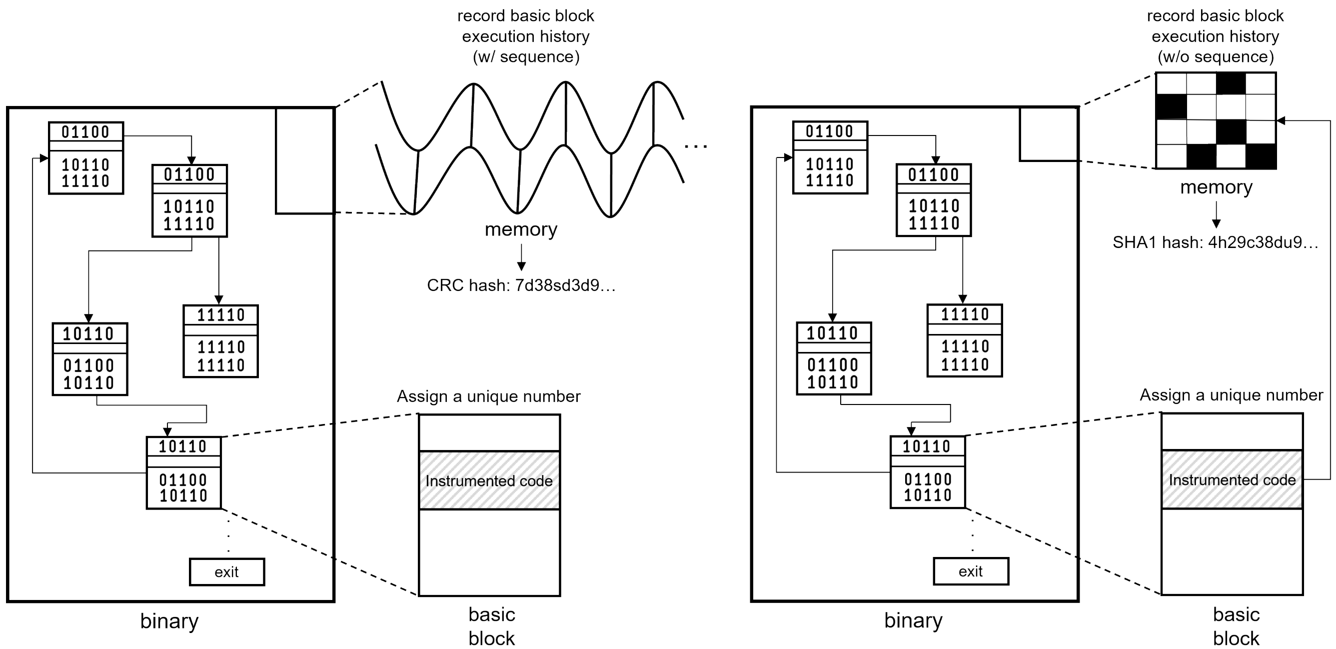

FIGURE 2. Internal design of EXHGen. Left and right side of the figure depicts version1 and version2 respectively.

program's execution flow has changed as shown in Fig. 1. Along with PoW-based verifiable computing, this function of the execution hash is applicable to diverse situations, such as detailed debugging work when problems arise in a deterministic program. It serves as a logical basis in verifiable computing and can be used to identify the problem if something goes wrong, for instance, when executing a deterministic program that must produce identical output for the same input. The execution hash can be used when debugging components, such as heap memory. The internal structure of heap memory greatly varies even with subtle differences in program execution, thus frequently leading to different results even from identical executions. Thus, the execution hash can be used to provide logical and reliable information about the program to all users, and it can be applied in certain situations to solve problems.

B. EXHGen_VERSION1

Fig. 2_version1 shows the overall design of version1. When building the test target program with the clang tool of version1, a unique number is assigned to a basic block in the target program's binary, and additional code is inserted at the same time. Afterward, when a block of a specific number is executed, the execution flow is recorded in order by the inserted code, and immediately before the program is terminated, the accumulated execution flow is calculated as the

execution hash value through the CRC (Cyclic Redundancy Check) algorithm. The basic block's execution record measured in version1 is typically expressed as code coverage. In version1, the code coverage is not tracked as an aggregated result, such as a hash map (e.g., conventional AFL code coverage) or counter data structure used in libfuzzer [21]. For precise observation, all execution records considering the order of the entire basic block visit record are observed, and the result is finally hashed. Thus, although it demands extensive load in terms of performance, version1 can reliably track subtly different execution flows that existing coverage measurement tools might miss. One may think that if all basic block execution flows are tracked, then memory usage will excessively increase if the program execution flow continues indefinitely, thus depleting memory. However, this can be solved with a trade-off of speed-memory usage. Whenever a new basic block is executed, rather than accumulating the record in memory, the existing hash is only updated internally. Hence, version1 has O(1) space complexity.

C. EXHGen_VERSION2

Fig. 2_version2 shows the overall design of version2. Build the test target program using the clang tool of version 2 in the same way as version 1. Unlike version1 however, the execution flow of each basic block is not recorded in order, and only information on the execution frequency of edges between the basic blocks is recorded in the form of an array. As the total number of edges in the program is fixed, the array always has a constant size. As in version1, the array in which the execution flow information is stored calculated as the execution hash value using the SHA1 algorithm. Given that version2 does not consider the execution flow order, the hash value remains identical even if the execution order of edges in the program changes, unlike version1. As a result, although the observation is not precise like version1, the time to measure the execution hash is always constant (e.g., O(1) time complexity) even if the execution flow becomes indefinitely long, as the array size and the length of the data itself are fixed. To conclude, this study proposes two tools with different application targets: for version1, a simple program that requires precise measurement, and for version2, a complex program that requires greater consideration of performance and time.

IV. APPLICATION

A. VERIFIABLE COMPUTING AND DEBUGGING

This section proposes two applications for this research: verifiable computing and debugging.

Suppose, a client outsources the computation of some functions to some untrusted clients. The clients that were assigned the work provide a result with proof that the work was correctly performed; hence, the client that outsourced the work should be able to judge the accuracy of the returned result. The proposed methodology can be applied in this case to prove that certain results were correctly computed in cases of verifiable computing that do not use TEE and TPM under the premise of PoW. For example, we propose the SETI@home project as an applicable scenario. SETI@home is a project that utilizes large-scale distributed computing technology to explore radio signals from space. It is a project to search for radio signals of extraterrestrial civilization by analyzing signals in the frequency band of planets that are likely to have intelligent life; using large-scale distributed computing technology. The master node outsources the radio signal analysis job to each node participating in the project and gathers the result. Each node participating in the job must use correct algorithm given by master — this must be verified if the participating nodes are untrusted. In general, distributed computing utilize TEE or TPM as root of trust to ensure code integrity. However, hardware support is not available to all systems, and additionally, untampered code do not always guarantee high level sementic of data processing that can subtly change due to runtime/external events such as interrupts. Although a system do not support hardware based root of trust, EXHGen can achieve same objective with high accuracy. For example, when the master node outsources the signal to analyze, EXHGen can effectively detect if the node is performing the job as-is without any tampering (intentionally or unintentionally) by comparing the excution hash value of other nodes. We can also apply EXHGen in a similar way to other distributed computing projects that require additional verification.

The second application is debugging. In the program development stage, debugging is conducted to find logical errors or bugs in the code, reveal the cause, and solve the problem. When debugging with repeated executions, if the results vary due to changes in the execution flow, then it may be impossible to debug the program effectively. Accordingly, researchers have continuously investigated techniques to make multi-threaded programs with frequent non-deterministic errors deterministic to debug them accurately. However, even the single-threaded programs (integrity protected via RoT) in this study produce non-deterministic errors due to asynchronous interrupts; and their frequency increases as the program grows in complexity. Therefore, the tool can be used to check whether the program's execution flow has changed, and changes according to system calls and various external factors can be modified through the proposed solution of function hooking. Through this, a program can be fixed to show the same execution flow when repeatedly executed, allowing the debugger to locate critical errors or bugs.

The proposed tool thus provides verifiable and reliable information in distributed computing environments and has various applications, such as creating an accurate debugging environment in the development stage.

B. NON-DETERMINISM

To describe the overall study design, first, EXHGen was applied to build binutils and coreutils programs, the study targets, and each of these built binaries was repeatedly executed

by applying a variety of execution options. Cases where the execution hash value changed during repeated execution were recorded, which were then statically and dynamically analyzed to confirm whether the difference was due to a non-deterministic execution flow. If the change was caused by an exceptional operation, such as an API/system call, it was modified through hooking to show a deterministic execution result. For this work, an additional test code and hooking code were written. For the test code, the test target program was repeatedly executed, and cases where hash values cause non-deterministic errors were automatically detected and managed. For the hooking code, libc APIs were appropriately modified based on the cause analysis, and non-deterministic execution was made deterministic. Non-deterministic execution covered in this study refers to cases where a single thread's basic block execution record varies with the heap state, system calls, and exceptional circumstances when calling external library functions.

V. IMPLEMENTATION

A. EXHGen_VERSION1

Fig. 2_version1 is implemented by adding a pass to the LLVM 6.0 as a sanitizer interface (`-fsanitize=`). In other words, a version of clang was created that adds a sanitizer, which performs the instrumentation desired, and a compiler option was added to that clang to build the source code of the target program with the additional code. We added LLVM pass code to following call back function: `runOnBasicBlock`. `runOnBasicBlock` is a call back function invoked by LLVM while parsing each basic block. In `runOnBasicBlock`, the pass brings the last location of the basic block using the `getTerminator` function to specify where to insert the instrumented code. The instrumented code simply invokes the custom function (version1_Trace) to trace the execution flow. In version1_Trace function, the unique basic block was identified based on return address (`__builtin_return_address`). `__builtin_return_address` is a special clang built-in function to get the return address of currently executing function. Since this address contains a code location inside the target basic block, this number can be used as a unique ID of the block. In this way, entire sequence of basic block IDs can be gathered and each time the ID is obtained, the internal CRC state can be updated. As a result, the final CRC state at the end of program execution contains the overall program execution flow. Version1 installs an exit handler (`atExit`) to print out the final CRC hash when the program is terminated. File interface is used to print out this information; therefore, when executing a test program built with version1, the working directory must be in a writable state.

B. EXHGen_VERSION2

Fig. 2_version2 is a program execution flow measurement tool created by modifying a part of the libfuzzer in the clang compiler and inserting additional code. The functions of version2 are implemented by inserting additional codes written in C++, and the most important code is the part where the execution hash is obtained. Libfuzzer executes the program to be implemented by dividing it into compartments referred to as module. Based on this, version2 measures the program execution hash of each module. The module consists of a basic block, which allows the program execution flow to be measured in units of modules. For this, version2 introduces an inline counter, which records the frequency of execution of the edge. Since the module consists of basic blocks, several basic blocks connected by the edge are executed when the module is called. In this case, the edge represents a jump between the basic blocks, which means that the information on the edge flow may be regarded as information on the execution frequency of the basic block. To obtain information on the final flow of the program, inline counter information for each module must be recorded. Accordingly, version2 inserts an array of up to eight bits to store an inline counter into each of the modules constituting the program. As the number of modules in the program and the number of basic blocks constituting the module do not change, the size of each inline counter array is also fixed. In this study, the inline counter array inserted into the module is regarded as the coverage of each module and hashed using the SHA1 algorithm. The operation is started for all modules constituting the program using a function of determining the start position and the end position of the module. The hash of each module is XOR-calculated to each other, and a hash chain is formed through this process. After the operation for all hashes is complete, the finally generated hash is dumped into the file. As a result, the hash finally dumped indicates an operation result in which information of all modules constituting the program is associated. This means that a very slight change in the flow of execution can bring a big change to the hash; hence, version2 can be used from various perspectives.

VI. EVALUATION

The tests were carried out in the following environment:

1) Ubuntu Linux 16.04 and 18.04 systems with 16 core CPU, 16 GB RAM, and 1TB SSD
2) 192 coreutils commands and corresponding options, 102 binutils commands and corresponding options, and 32 OSS-Fuzz programs

A. NON DETERMINISM TEST

Non-determinism tests were repeatedly performed on the selected commands and programs to examine whether the result varied due to any other secondary factor under a single-threaded situation. This was done to demonstrate the meaningfulness of finding non-deterministic execution flows in single-threaded commands and programs, which this study investigated, rather than multi-threaded ones, as in many previous studies. Each binutils/coreutils command was repeatedly executed 100,000 or 1 million times depending on

TABLE 1. Number of execution hash deviation in OSS-Fuzz evaluation. We iterated same execution and counted deviated hash as error.

Application	errors	Application	errors
libssh2	5/93,500K	proxygen	0/65,800K
freeimage	6/48,200K	ffmpeg	0 / 5,530K
picotls	7/36,500K	lame	5 /15,700K
xvid	2/64,140K	libtiff	7 / 6,360K
arrow	5/23,150K	bzip2	12 / 15,190K
binutils	5/22,330K	giflib	3 / 6,180K
zlib	11/67,740K	capstone	1/17,300K
lz4	10/67,990K	libmpeg2	0/970K
php	2/23,920K	libzip	2/21,800K
libpng-proto	0/30,600K	flac	4/17,480K
gfwx	7/34,200K	hostap	0/96,000K
clamav	7/22,760K	wavpack	7/12,930K
eigen	7/24,430K	c-ares	2/14,570K
jansson	18/23,140K	htslib	0/10,250K
matio	12/19,000K	lodepng	0/18,000K
avahi	19/20,260K	openssl	0/9,000K

performance to find non-deterministic cases. To catch non-deterministic case, a python script compares execution hashes gathered from multiple executions. EXHGen_version1 was applied with the assumption that the hash value obtained by the first execution was the correct value. Command execution was automated, and the hash values for the program execution flow accumulated in the file for each execution were compared. When the values differed, it was judged as a non-deterministic error. In Table 3, Err indicates the number of non-deterministic errors in the two commands. While most of the commands exhibited a deterministic program execution flow, the -O option of the strip-new command in binutils showed relatively many errors (15). This is because, unlike other commands, intermittent function calls related to system calls were omitted due to repeated execution.

The same algorithm was used in OSS-Fuzz; the number of iterations was manually based on the test target program's execution speed; 32 programs were tested about 1 million times each. Since EXHGen_version2 was used, every time a program was executed, a file containing the hash value generated according to the number of basic block executions was generated. The md5 hash values for these files were obtained to check whether the execution flow changed; if the hash values were different, then it was judged as a non-deterministic error. Table 1 shows the error frequency in each program. The results in Table 1 and Table 3 indicate that the error rate rises with the program's complexity due to the variety of variables in repeated execution. Most of the 32 samples showed a significant error rate, while only six did not show any error.

B. PERFORMANCE TEST

Since additional code was added in EXHGen_version1 to calculate the hash value for the program execution flow, it was predicted that the execution time would slow down as more codes were physically executed like Table 2. To examine how much the execution time slowed and how much the time performance differed with the existing executable file analyzer, the time performance was measured with repeated executions of binutils/coreutils commands Table 3. Due to page limitation, Table 3 summarizes subset of our data set. Full data is available in Table 4, Table 5, and Table 6. As binutils/coreutils considers even the order to calculate the hash for the execution history, it is meaningful to compare the time between the method using EXHGen_version1 and the method not using it. However, for EXHGen_version2 used in OSS-Fuzz, since the hash is calculated only once regardless of the program's execution time, the hash calculation time was not additionally analyzed. In Table 3, the number of iterations was fixed for all cases to 100,000 and measured the difference in the time performance of each command and option depending on whether EXHGen_version1 was applied in binutils/coreutils.

According to evaluation, in almost all cases, execution took longer when using EXHGen_version1, though the difference was not large. version1 can also be implemented using a DBI tool, such as Intel pin tool; therefore, we additionally implemented EXHGen based on Intel pin's trace feature (denoted as D.T in the table) and the performance (consumed time memory) was accordingly compared. Compared to C.T, D.T clearly showed a significant overhead. In terms of time taken to measure the execution flow, using EXHGen_version1 was 1% to 10% slower than not using any tool at all, whereas using Intel pin tool was nearly 10 times slower. This indicates that EXHGen can measure a program's execution flow more quickly and efficiently.

C. NON-DETERMINISM FIX

Given that non-deterministic errors were found even in single-threaded operations in the above non-determinism test, a non-determinism fix test was performed to demonstrate that it is meaningful to resolve the error and make the execution flow deterministic. For binutils/coreutils commands, where errors were found by comparing the execution hash values, `ltrace` and `strace` were run to find the cause of the error. This was done by inserting ltrace/strace code into the above code that was repeatedly executed. From the results, the causes were classified into four categories:

- error inevitably occurred during repetition because the operation was a one-time command
- error occurred because the disk and memory conditions changed in real time
- a problem with the code used in the test
- a specific reason could not be identified

Particularly, it was difficult to measure the determinism of the execution flow accurately when an error occurred due to the OS used for the test, computer specification or environment, disk environment, among others. As such, of these four categories, an additional test was performed on commands for which the cause of error was unknown.

The test was conducted on cases of the -O option of the `strip-new` command in binutils, which revealed the 15 errors as shown in Table 3. Unknown errors arose for this command with a probability of about 0.001%. The execution was repeated approximately 450,000 times for a more

TABLE 2. Size comparison of code by `size` command. O.S stands for original code size without hash generation, C.S stands for changed code size with EXHGen.

App	O.S	C.S	Delta	App	O.S	C.S	Delta	App	O.S	C.S	Delta
b2sum	46,777	106,683	43.8%	basename	24,620	50,184	49.1%	id	33,371	69,100	48.3%
cksum	26,954	53,501	50.4%	chgrp	55,909	116,275	48.1%	hostid	23,471	47,998	48.9%
base32	32,043	64,752	49.5%	chown	58,240	119,949	48.6%	md5sum	36,568	74,537	49.1%
base64	31,707	63,480	49.9%	cp	102,318	203,085	50.4%	who	42,805	146,012	29.3%
nproc	27,474	53,802	51.1%	dircolors	35,561	66,864	53.2%	whoami	23,759	48,473	49.0%
split	47,207	93,937	50.3%	expr	109,037	246,285	44.3%	wc	36,924	75,183	49.1%
cat	28,164	57,918	48.6%	mv	109,458	229,324	47.7%	stat	74,613	230,141	32.4%
mkdir	44,455	157,491	28.2%	touch	80,329	324,608	24.7%	tac	98,691	231,344	42.7%
pwd	26,626	57,790	46.1%	join	42,770	79,786	53.6%	uniq	35,430	66,952	52.9%
df	78,047	227,815	34.3%	nl	101,855	230,722	44.1%	stty	65,461	176,399	37.1%
dir	124,716	255,094	48.9%	test	41,496	152,589	27.2%	tty	23,032	47,163	48.8%
dirname	23,853	48,741	48.9%	fmt	33,457	63,969	52.3%	sync	24,982	51,499	48.5%
echo	25,706	50,807	50.6%	unexpand	28,921	57,651	50.2%	uptime	39,733	90,558	43.9%
env	35,266	68,528	51.5%	ls	124,716	255,094	48.9%	tsort	29,222	58,736	49.8%
head	35,400	70,087	50.5%	ar	1,135,978	2,339,510	48.6%	printenv	23,451	48,027	48.8%
uname	24,996	49,954	50.0%	nm-new	1,122,886	2,314,978	48.5%	objcopy	1,244,529	2,559,786	48.6%
users	25,530	50,841	50.2%	objdump	2,327,712	4,115,481	56.6%	strip-new	1,244,529	2,559,786	48.6%
ptx	131,743	282,427	46.6%	readelf	962,561	1,836,220	52.4%	cxxfilt	1,103,313	2,279,220	48.4%
pinky	30,512	58,617	52.1%	size	1,106,797	2,285,223	48.4%	elfedit	30,147	54,718	55.1%
basenc	41,051	82,099	50.0%	strings	1,110,145	2,293,751	48.4%	ranlib	1,135,978	2,339,510	48.6%

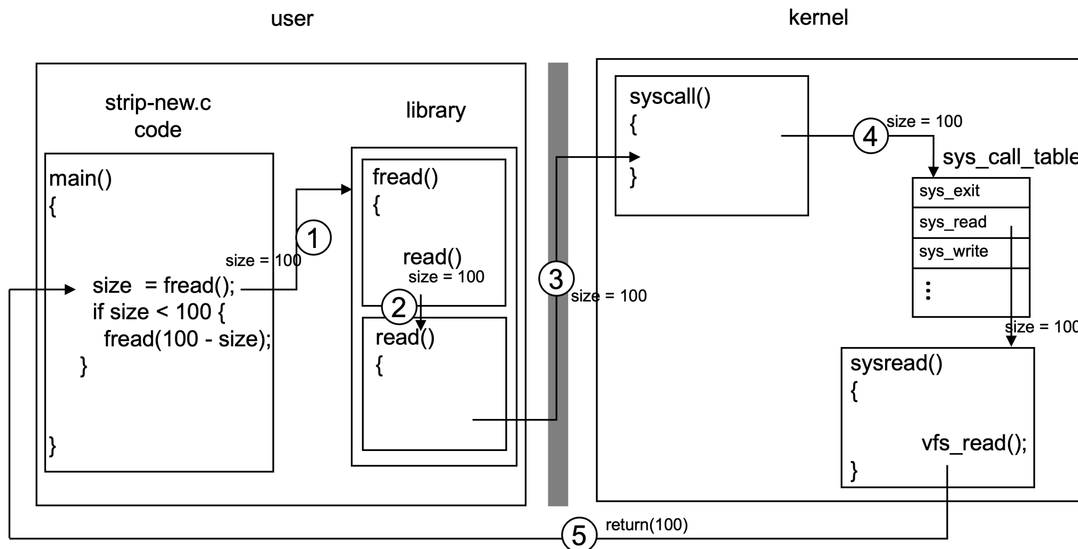

FIGURE 3. Normal execution case. `fread` retrieves total 100 bytes in a single trial.

detailed observation, in which case the same hash value error occurred six times and the probability slightly rose to approximately 0.0012%. According to the normal execution flows shown in Fig. 3 and the execution flows where error occurred shown in Fig. 4 using ltrace, although most showed the same results, it was observed that when a non-deterministic situation occurred, the fread function call was intermittently omitted due to a system call error. Consequently, it was not possible to read the full data from the file as expressed in No. 5 of Fig. 4; therefore, fread function hooking code (myfread.c) was written to fix the execution flow deterministically, which set the reason for non-determinism.

The point of this hooking code Algorithm 1 is to fix the execution flow so that when reading a file using the fread function, the contents corresponding to the size passed as an argument are read accurately for every execution without exception. This code was composed in a format similar to the existing fread function and inserted the code `original_fread = dlsym(RTLD_NEXT, "fread")` to declare the original fread function as original_fread. Code was then written to check whether the returned value (total_count) when passing the same parameters to the original_fread function and calling it was identical to the count parameter value passed when calling the fread function. If the values were different, then the code proceeded through Nos. 6–10 shown in Fig. 4, determined that the function returned while reading the file, and read the address after the interrupted address again with the original_fread function. This was implemented with the code `original_fread( (void*)((unsigned int)ptr + total_count*size), size, count - total_count, stream)`. Thus, it was modified so that

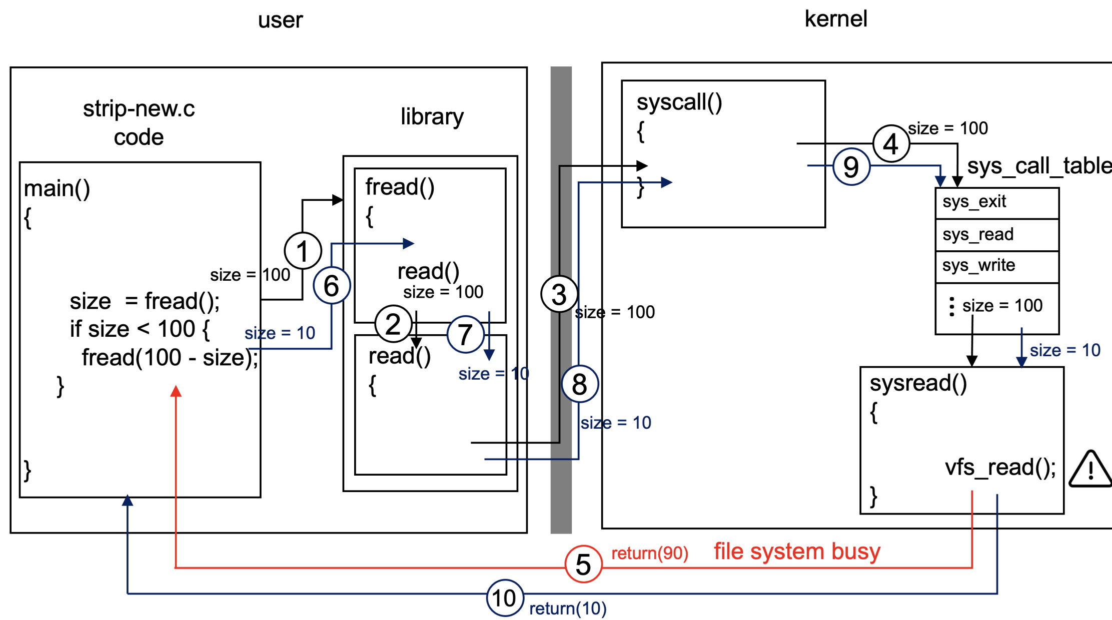

FIGURE 4. **Abnormal execution case.** `fread` **retrieves total 100 bytes in two trials** `10` **and** `90` **respectively.**

the hooking library could handle the exception-handling logic that could continuously read the contents of the file until count and total_count were equal. When count and total_count were equal, total_count was returned and the function was terminated.

The completed code myfread.c was compiled as a shared library using the command `gcc -Wall -fPIC -shared -o myfread.so myfread.c -ldl`, after which the command `LD_PRELOAD=./myfread.so./ strip-new -O binary test` was executed so that when strip-new was executed, the myfread.so shared library could be loaded and the hooked fread function called. Finally, ltrace was used to check whether the hooked fread function was properly called; an error rate of 0% was observed when the test was repeated 1,000,000 times. In conclusion, the desired deterministic program execution flow result was successfully derived.

Algorithm 1: patched Libc Fread Function.

Data: `PTR`, `size`
Result: the final value of `total_size`
`total_size` = 0;
`original_fread` → `dlsym`(“`fread`”);
`remaining_size` = `size`;
while *`total_size`* != *`size`* **do**
  `current_size` = `original_fread`(PTR + total_size, remaining_size);
  `total_size` += `current_size`;
  `remaining_size` −= `current_count`;
return `total_size`;

VII. RELATED WORK

A. PROGRAM TRACE

Tools for tracing programs [29]–[31] often find detailed elements, such as program history (e.g., function calls), thread operation methods, and various event types in the program execution stage. Since the execution hash can determine whether the execution flow changed when tracing the program, it can effectively measure performance to judge whether the program was executed deterministically. SROH [29] calculates the traced hashes for values stored in memory corresponding to the random read and write access of the program as proposed in OH [30], thus automatically detecting non-deterministic program areas and protecting integrity. The greatest difference in the current study is that the determinism of the program itself can be confirmed because the entire basic block call history is collected to create a table of the call frequency, from which the hash is calculated.

B. VERIFIABLE COMPUTING

As the execution hash in this study is designed to verify whether a program is executed correctly, it can be used for verifiable computing. Many prior studies on verifiable computing dealt with situations where the input-output relationship is clear, such as in cryptographic operations. [32] presented a method that defends against threats, such as dishonesty from untrusted clients and enables them to prove that they accurately performed most of the work with a high probability. In this process, although the act of confirming whether a specific operation result was returned is similar

TABLE 3. Evaluation result using binutils/coreutils. The numbers are measured based on 100K iterative executions. Opt is the parameter given to the application. * indicates that application executes without parameter. Err is count of execution hash deviation. O.T stands for original execution time without hash generation, C.T stands for changed execution time with EXHGen and D.T stands for execution time with DBI based hash generation.

App	Opt	Err	O.T	C.T	D.T	App	Opt	Err	O.T	C.T	D.T
b2sum	*	0	2m8.3s	2m18.4s	124m27.5s	basenc	–base64	0	2m11.6s	2m34.6s	73m49.8s
	-b	0	2m14.6s	2m6.1s	84m44s	chgrp	*	1	4m13.4s	4m31.5s	108m27.7s
cksum	*	0	2m57.8s	2m5.8s	93m41.8s		-R	0	4m21.8s	4m35.5s	108m59s
base32	*	0	3m14.6s	2m9s	89m38.6s	chown	*	0	4m19.6s	4m38.6s	119m21.6s
	-d	0	3m19.1s	2m6.5s	79m48.3s		-R	0	4m25.8s	4m30.3s	148m9.2s
base64	-d	0	3m36.9s	2m11s	60m35.4s	cp	*	0	11m31s	12m12.4s	77m18.6s
basename	*	0	3m16.5s	2m5s	45m26.7s		-a	0	11m31.1s	12m10.3s	79m29.2s
	-s	0	3m25.3s	2m11.9s	50m59.9s	dircolors	*	0	1m33.8s	2m1.4s	69m39.2s
split	*	0	37m51.2s	39m59.1s	66m39.5s	expr	*	0	3m47.3s	3m49.2s	80m48.9s
	-a	0	37m15.8s	39m14.6s	64m19.5s	mv	*	0	16m8.2s	16m33.1s	74m3.2s
cat	*	0	3m18.8s	2m5.8s	39m31.6s		-b	0	16m8.6s	16m42.5s	77m13.3s
	-A	0	3m20.6s	3m14.3s	46m22s	touch	*	0	1m49.1s	1m52.3s	61m17.7s
mkdir	*	0	33m57.6s	37m18.6s	60m6.3s		-a	0	1m55.8s	2m0.1s	61m11.5s
	-m	0	34m29.4s	37m9.7s	39m36.4s	join	*	0	16m27.9s	20m31.1s	75m26.2s
pwd	*	0	1m51.9s	1m55.5s	40m51s		-a	0	16m29.7s	20m38.9s	76m34.1s
	-L	0	1m55s	1m58.1s	39m56.8s	nl	*	0	2m21.3s	2m37.8s	67m51.6s
df	*	0	3m32.9s	3m25.8s	122m27.1s		-b	0	2m22.2s	2m39.9s	65m18.1s
	-a	0	3m41.7s	3m48.8s	76m5.7s	test	*	0	2m40.2s	1m57.8s	38m11.5s
dir	*	0	24m18.3s	36m59.8s	71m17.5s	fmt	*	0	2m59.4s	4m56.1s	75m37.9s
dirname	*	0	32m57.1s	37m14.1s	59m47.5s		-t	0	2m59.1s	4m47.2s	61m28.2s
echo	*	0	33m7s	36m58.4s	47m24s	unexpand	*	0	2m29.3s	6m12.3s	45m42.3s
	-n	0	33m2.2s	36m21.1s	76m18.9s	ls	*	0	3m18.4s	4m47.3s	52m3s
env	*	0	1m50.7s	1m51.4s	49m39.7s		-a	0	3m19.1s	4m48.1s	53m31.2s
	-i	0	2m11.8s	2m18.7s	75m31s	nproc	*	0	1m40.4s	1m44.7s	47m22.6s
head	*	0	17m17.8s	28m3.9s	45m52s		–all	0	1m46.6s	2m22.1s	49m12.1s
	-c	0	18m59s	28m8.8s	104m25.8s	tac	*	0	1m50.9s	2m2.1s	47m40.6s
id	*	0	1m57.9s	1m59.7s	88m13.2s	uniq	*	0	1m25.5s	1m38s	50m57.4s
hostid	*	0	1m14.3s	2m20.7s	78m16.4s	stty	*	0	1m33.1s	1m49.7s	60m30.3s
md5sum	*	0	2m15.8s	6m3.4s	68m43s	tty	*	0	1m36.6s	2m5.3s	46m19.3s
	-b	0	2m29.4s	6m5.6s	88m48s	sync	*	0	9m41.4s	17m33.9s	45m29.3s
who	*	0	3m55.1s	7m5.4s	76m1.1s	uptime	*	0	1m42.5s	1m57.4s	67m47.4s
whoami	*	0	2m40.6s	3m9s	71m37.5s	tsort	*	0	1m3.6s	1m21.5s	43m49.8s
wc	*	0	3m35.5s	9m55.6s	51m57.1s	uname	*	0	1m1.9s	1m13.8s	55m9.6s
	-l	0	1m36.6s	1m52.5s	80m25s		-s	0	1m1.8s	1m18.4s	45m28.4s
stat	*	0	3m16.8s	3m45.2s	123m32.7s	users	*	0	1m57.1s	2m11.7s	51m36.5s
	-t	0	1m33.8s	1m45.9s	77m30.6s	ptx	*	0	1m10.5s	1m29.1s	65m5.2s
printenv	*	0	1m32.3s	1m47.9s	46m8.7s		-A	0	1m10.8s	1m29s	66m57s
nm-new	-a	0	1m30.5s	2m7.4s	169m54.3s	pinky	*	0	1m37.3s	1m45.2s	118m49.6s
	-A	0	1m33.7s	2m9s	168m52.4s		-l	0	1m10.8s	1m24.9s	71m51.6s
objdump	-a	0	1m23.8s	1m36.5s	115m55.2s	ar	-p	0	1m25.8s	1m38.7s	76m12s
	-f	0	1m24.6s	1m28.6s	117m58.5s		-d	0	3m10.6s	3m34.4s	73m2.9s
readelf	-a	0	1m29s	1m47.7s	142m15.7s	cxxfilt	-i	0	1m13.7s	1m24.3s	46m41.1s
	-h	0	1m21.8s	1m41.5s	114m9.4s		-p	0	2m17.5s	2m18.4s	46m40s
size	-A	0	1m28.3s	1m44.7s	87m1.7s	elfedit	–output-mach	0	4m25.4s	4m10.8s	59m22.8s
strings	*	0	1m19.6s	1m37s	66m11s	ranlib	-U	0	5m0.7s	5m36.3s	111m26s
	-a	0	1m20.3s	1m38.7s	66m23.8s		-D	0	5m2.5s	5m37.1s	110m58.5s
objcopy	-w	0	11m37.3s	15m58.1s	113m20.4s	strip-new	-I	0	5m25.4s	5m38s	104m30.3s
	-x	22	6m35.2s	17m5.4s	115m3.5s		-O	15	4m53.4s	5m18.3s	105m51s

to the current study, a difference is that it uses mathematical operations.

Furthermore, although [33]–[35] have similar objectives to this study, their approaches to solutions slightly differ in terms of mathematical and cryptographic operations. The current study applied verifiable computing to trace the execution flow of programs and demonstrated that the proposed methods can verify operations and identify whether computations were properly executed. Methods that can implement verifiable computing include PoW, TEE, and TPM. [25], [36]–[39] investigated ways to ensure the integrity of work and programs by utilizing these methods. PoW-based verifiable computing, the application target of this study, was covered in [25], which investigated efficient statistical techniques using PoW for cloud and fog computing and proposed a method that provides secure and transparent transactions by verifying data blocks while solving PoW's limitations of high resource consumption and long working time. To improve computing security, [36]–[39] used hardware-based methods applying TEE and TPM, which ensure program integrity. The studies referenced in [36] aimed to provide a secure and reliable distributed computing environment, same as our study.

TABLE 4. Full data set of binutils&coreutils evaluation.

Binutils											
App	**Opt**	**Err**	**O.T**	**C.T**	**D.T**	**App**	**Opt**	**Err**	**O.T**	**C.T**	**D.T**
nm-new	*	0	1m29.6s	2m3.2s	168m5.3s	strings	*	0	1m19.6s	1m37s	66m11s
	-a	0	1m30.5s	2m7.4s	169m54.3s		-a	0	1m20.3s	1m38.7s	66m23.8s
	-A	0	1m33.7s	2m9s	168m52.4s		-d	0	1m26.5s	1m23.7s	77m21.5s
	-B	0	1m32.6s	2m4.3s	168m48.4s		-f	0	1m23s	1m45s	61m31.3s
	-C,	0	1m35.7s	2m22.9s	170m21.7s		-n	0	1m21.5s	1m21.6s	61m17.7s
	-D	0	1m21.3s	1m34.2s	158m48.7s		-<number>	0	1m16.1s	1m24.4s	61m20.5s
	-g	0	1m26.2s	1m46.9s	146m16.8s		-t	0	1m17.9s	1m38.2s	62m51.4s
	-l	0	6m3.8s	11m44.8s	151m49.4s		-T	0	1m31.8s	1m53.2s	65m33.5s
	-n	0	1m34.4s	2m5.6s	132m23.4s		-e	0	1m26.1s	1m47.2s	65m43.7s
	-r	0	1m37.7s	2m6.3s	132m35.8s	objcopy	*	0	2m48.3s	3m7.6s	112m14.6s
	-S	0	1m27.2s	2m9s	135m41.7s		-I	0	17m57.7s	10m36.9s	113m24s
	-s	0	1m33.8s	1m56.7s	116m18.2s		-O	0	14m51.9s	15m34.2s	113m26.8s
	–quiet	0	1m22.9s	1m37.6s	135m47.3s		-F	0	13m	6m12.3s	115m13.6s
	-u	0	1m22.8s	1m45.6s	123m54.6s		-p	0	9m14.1s	16m26.3s	115m33.1s
objdump	-a	0	1m23.8s	1m36.5s	115m55.2s		-D	0	9m40.9s	16m25.5s	111m17.8s
	-f	0	1m24.6s	1m28.6s	117m58.5s		-S	0	4m1.8s	10m22.8s	108m47.7s
	-p	0	1m21s	1m30.5s	142m4.5s		-g	5	6m20.1s	14m16.1s	115m16.4s
	-h	0	1m26.6s	1m31.9s	138m10.7s		–strip-dwo	10	10m23.7s	11m3.2s	120m27.9s
	-x	0	1m25.3s	1m36s	148m55.5s		-K	0	11m24.5s	13m51.8s	122m2.9s
	-d	0	1m28.5s	1m42.4s	170m53.1s		-L	0	12m17.5s	22m11.1s	117m4.7s
	-D	0	1m47s	1m47.2s	178m21.8s		–weaken	0	11m19.8s	21m45.6s	115m0s
	-S	0	1m30s	1m44.8s	173m39.3s		-w	0	11m37.3s	15m58.1s	113m20.4s
	-g	0	1m45.2s	1m50.1s	145m28.5s		-x	22	6m35.2s	17m5.4s	115m3.5s
	-e	0	1m39.6s	1m41s	147m42.2s		-X	28	12m23.3s	17m5.4s	111m17.2s
	-G	0	1m33.2s	1m49.3s	127m41.2s	ar	-d	0	3m10.6s	3m34.4s	73m2.9s
	-W	0	1m46.6s	1m51s	151m32s		-m[ab]	0	1m21.9s	1m37.3s	72m57.3s
	-T	0	1m32.4s	1m49.1s	138m50.7s		-p	0	1m25.8s	1m38.7s	76m12s
	-r	0	1m34s	1m50.8s	134m34.9s		-q[f]	0	1m24.6s	1m38.5s	106m34.6s
	-R	0	1m38.8s	1m50.2s	140m42.6s		-r[ab][f][u]	0	3m24.2s	3m37.5s	119m45.6s
readelf	-a	0	1m29s	1m47.7s	142m15.7s		-s	0	1m32.1s	1m43.2s	103m48.5s
	-h	0	1m21.8s	1m41.5s	114m9.4s		-t[O][v]	0	1m23.8s	1m37s	75m39.5s
	-l	0	1m22.8s	1m42.3s	112m6.9s		-x[o]	0	1m43.1s	1m55.9s	79m1s
	-S	0	1m31s	1m41.3s	105m36.4s	cxxfilt	*	0	2m15.3s	2m19.5s	47m21.1s
	-g	0	1m27.4s	1m39.1s	97m1.6s		-p	0	2m17.5s	2m18.4s	46m40s
	-t	0	1m33.4s	1m45.6s	97m10.2s		-i	0	1m13.7s	1m24.3s	46m41.1s
	-e	0	1m34.1s	1m43.6s	100m41.9s		-_	0	1m45.9s	1m58.2s	46m48.3s
	-s	0	1m29.6s	1m42.1s	97m14s	elfedit	–output-mach	0	4m25.4s	4m10.8s	59m22.8s
	–dyn-syms	0	1m28.1s	1m41.4s	97m10.4s		–output-type	0	4m6.6s	3m43.9s	61m22.6s
	-L	0	1m28.5s	1m39.5s	110m14.4s	ranlib	*	0	5m4.6s	5m33.7s	111m11.8s
	-x	0	1m24.5s	1m35.6s	90m5.7s		-D	0	5m2.5s	5m37.1s	110m58.5s
	-p	0	1m26.6s	1m37.5s	89m54s		-U	0	5m0.7s	5m36.3s	111m26s
	-I	0	1m26.1s	1m40.3s	85m58.6s		-t	0	4m49.1s	5m8.3s	87m25.1s
size	*	0	1m25.1s	1m41.9s	85m44.8s	strip-new	-O	15	4m53.4s	5m18.3s	105m51s
	-A	0	1m28.3s	1m44.7s	87m1.7s		-I	0	5m25.4s	5m38s	104m30.3s
	-B	0	1m30.4s	1m27.1s	86m55.7s		-F	0	5m18.2s	5m7.6s	104m4.5s
	-o	0	1m25.5s	1m39.6s	86m35.7s		-p	0	5m22.4s	5m40.6s	103m59.9s
	-d	0	1m28.2s	1m44.2s	86m47.4s		-D	0	5m8.1s	5m22.5s	102m22.3s
	-x	0	1m27.2s	1m42s	86m22.4s		-R	0	4m31.3s	4m41s	102m47.7s
Coreutils											
b2sum	*	0	2m8.3s	2m18.4s	124m27.5s	whoami	*	0	2m40.6s	3m9s	71m37.5s
	-b	0	2m14.6s	2m6.1s	84m44s	wc	*	0	3m35.5s	9m55.6s	51m57.1s
	-c	0	2m22.8s	2m5.9s	127m30.3s		-l	0	1m36.6s	1m52.5s	80m25s
	-I	0	2m24.9s	2m4.7s	95m24.5s		-w	0	2m27.4s	7m53.7s	105m43.7s
	–tag	0	2m33.2s	2m13.7s	60m2.7s		-c	0	1m35.1s	1m50s	78m55.7s
	-t	0	2m29.4s	2m9.9s	89m6.6s	stat	*	0	3m16.8s	3m45.2s	123m32.7s
	-z	0	2m35.1s	2m2.9s	60m46.6s		-t	0	1m33.8s	1m45.9s	77m30.6s
	–status	0	2m25.1s	2m20.2s	64m16.5s		-f	0	1m37.9s	1m52.6s	95m12.2s
	-w	0	2m37.1s	2m11.7s	62m44.5s		-c	0	1m37s	1m51.4s	74m44.9s
cksum	*	0	2m57.8s	2m5.8s	93m41.8s	printenv	*	0	1m32.3s	1m47.9s	46m8.7s
base32	*	0	3m14.6s	2m9s	89m38.6s	basenc	–base64	0	2m11.6s	2m34.6s	73m49.8s
	-d	0	3m19.1s	2m6.5s	79m48.3s		–base32	0	1m10.2s	1m17.3s	74m31.5s
	-I	0	3m23.5s	2m7.6s	72m48.9s		–base16	0	1m10.1s	1m20.9s	72m24.5s
	-w	0	3m28s	2m6.8s	57m18.7s		-d	0	1m13.5s	1m24.4s	69m59.5s
base64	*	0	3m34.1s	2m3.8s	46m21.3s		-i	0	1m15.6s	1m23.0s	82m33.8s
	-d	0	3m36.9s	2m11s	60m35.4s		-w	0	1m9.1s	1m15.5s	90m39.4s
	-I	0	3m41.6s	2m11.7s	47m13.6s	chgrp	*	1	4m13.4s	4m31.5s	108m27.7s
	-w	0	3m41.7s	2m10.2s	51m50.9s		-R	0	4m21.8s	4m35.5s	108m59s

TABLE 5. Full data set of coreutils evaluation.

Coreutils											
App	**Opt**	**Err**	**O.T**	**C.T**	**D.T**	**App**	**Opt**	**Err**	**O.T**	**C.T**	**D.T**
basename	*	0	3m16.5s	2m5s	45m26.7s		-v	0	4m22.7s	4m40.4s	135m1.4s
	-s	0	3m25.3s	2m11.9s	50n59.9s		-c	0	4m26.7s	4m29.6s	117m55.5s
	-a	0	3m16.5s	2m8.4s	62m57.3s	chown	*	0	4m19.6s	4m38.6s	119m21.6s
split	*	0	37m51.2s	39m59.1s	66m39.5s		-R	0	4m25.8s	4m30.3s	148m9.2s
	-a	0	37m15.8s	39m14.6s	64m19.5s		-v	0	4m27.3s	4m43.2s	145m12.2s
	-b	0	36m22.5s	38m29.8s	61m55.2s		-c	0	4m22.9s	4m14.1s	110m31.9s
	-d	0	38m48.2s	39m37.4s	60m59.2s	cp	*	0	11m31s	12m12.4s	77m18.6s
	-l	0	41m13.8s	40m26s	60m53.7s		-a	0	11m31.1s	12m10.3s	79m29.2s
	-n	0	41m9.8s	40m3.6s	75m42.5s		-p	0	9m45.7s	10m10.8s	78m38.2s
cat	*	0	3m18.8s	2m5.8s	39m31.6s		-i	0	8m35.8s	9m4.9s	81m3.7s
	-A	0	3m20.6s	3m14.3s	46m22s		-r	0	11m37.1s	12m7s	65m27.9s
	-b	0	3m21.6s	3m34.3s	43m59.6s		-v	0	8m51.5s	9m18.6s	71m0.2s
	-e	0	3m19.1s	3m15.5s	40m22s		-u	0	18m46.7s	19m13.4s	82m8.7s
	-n	0	3m18.5s	3m34.7s	43m53s		-b	0	15m45.5s	16m30s	100m20.7s
	-s	0	3m15.7s	3m31.9s	54m40.2s	dircolors	*	0	1m33.8s	2m1.4s	69m39.2s
	-t	0	3m8.9s	3m17.3s	43m25.9s	expr	*	0	3m47.3s	3m49.2s	80m48.9s
	-T	0	3m6.7s	3m28.9s	54m12.3s	mv	*	0	16m8.2s	16m33.1s	74m3.2s
	-v	0	2m57.4s	3m3.2s	43m43.4s		-b	0	16m8.6s	16m42.5s	77m13.3s
mkdir	*	0	33m57.6s	37m18.6s	60m6.3s	touch	*	0	1m49.1s	1m52.3s	61m17.7s
	-m	0	34m29.4s	37m9.7s	39m36.4s		-a	0	1m55.8s	2m0.1s	61m11.5s
	-p	0	34m21.7s	36m47.6s	39m14.6s		-c	0	1m57.7s	2m1.2s	59m47.4s
	-v	0	36m1s	37m32.1s	67m29.3s		-d	0	1m53.4s	2m0.8s	88m34.2s
pwd	*	0	1m51.9s	1m55.5s	40m51s		-m	0	1m55.5s	2m3.1s	67m33.4s
	-L	0	1m55s	1m58.1s	39m56.8s		-r	0	1m52.6s	1m56.9s	72m26.9s
	-P	0	1m55.1s	1m57.9s	41m11.4s		-t	0	1m57.6s	2m1.3s	71m39.5s
df	*	0	3m32.9s	3m25.8s	122m27.1s	join	*	0	16m27.9s	20m31.1s	75m26.2s
	-a	0	3m41.7s	3m48.8s	76m5.7s		-a	0	16m29.7s	20m38.9s	76m34.1s
	-B	0	2m44.2s	2m28.5s	78m21.9s		-e	0	16m47.5s	20m59.3s	77m19.7s
	–total	0	1m36.2s	3m23.2s	97m47.6s		-i	0	15m51s	20m8.9s	74m52.6s
	-h	0	3m11.4s	3m21.1s	75m31.2s	nl	*	0	2m21.3s	2m37.8s	67m51.6s
	-H	0	3m4.9s	3m23.8s	95m58.7s		-b	0	2m22.2s	2m39.9s	65m18.1s
	-i	0	2m3.6s	2m8.3s	75m21.2s		-d	0	2m21.6s	2m40.5s	64m19.3s
	-k	0	2m5.4s	2m9.3s	76m28.5s		-f	0	2m11.2s	2m29.4s	64m23s
	-P	0	2m53.2s	3m21.6s	75m45.3s		-h	0	2m7.1s	2m7.6s	62m31.3s
	-T	0	2m6.8s	2m4.6s	72m28.7s		-i	0	2m13.5s	2m29.9s	75m54.7s
dir	*	0	2m31.4s	2m38.2s	71m17.5s		-l	0	2m14s	2m30.3s	76m21.3s
	-a	0	2m37.2s	2m39.6s	51m57.2s		-n	0	2m9.3s	2m25.9s	80m38.2s
	–author	0	2m34.8s	2m40.2s	79m45.2s		-p	0	2m9.3s	2m25.9s	66m23.6s
	-b	0	2m40.4s	2m44.4s	80m38.5s	test	*	0	2m40.2s	1m57.8s	38m11.5s
	-c	0	2m42.7s	2m45.9s	92m20s	fmt	*	0	2m59.4s	4m56.1s	75m37.9s
	-d	0	2m2.7s	2m9.6s	88m1.7s		-t	0	2m41s	2m44.6s	61m28.2s
	-g	0	2m53s	2m 58.3s	94m12.3s		-w	0	3m4.9s	4m56.5s	71m59.5s
	-N	0	2m35.8s	2m38s	96m18.6s	unexpand	*	0	2m29.3s	6m12.3s	45m42.3s
	-u	0	2m37.8s	2m42.4s	90m21.8s	ls	*	0	3m18.4s	4m47.3s	52m3s
dirname	*	0	1m55.1s	1m59.6s	59m47.5s		-a	0	3m19.1s	4m48.1s	53m31.2s
echo	*	0	1m54.3s	2m0.4s	47m24s		-b	0	3m24.3s	4m53.4s	54m11.1s
	-n	0	1m56.9s	2m2.1s	76m18.9s		-c	0	4m28.2s	5m38.7s	55m43.8s
	-e	0	2m0.4s	2m5.4s	76m54.5s		-d	0	2m28.2s	2m58.2s	51m12.1s
env	*	0	1m50.7s	1m51.4s	49m39.7s		-D	0	3m31.8s	5m6.4s	54m23.4s
	-i	0	2m11.8s	2m18.7s	75m31s		-f	0	3m11.2s	4m36s	49m8.1s
	-u	0	1m9s	1m13s	80m8.5s		-g	0	7m4.8s	9m9s	103m50.3s
head	*	0	17m17.8s	28m3.9s	45m52s		-H	0	3m20s	4m51.3s	53m27.7s
	-c	0	18m59s	28m8.8s	104m25.8s		-i	0	4m50.8s	5m56.7s	55m9.2s
	-n	0	19m22.3s	28m1s	123m53.9s		-I	0	6m44.4s	9m32.8s	62m20s
	-q	0	19m21.2s	27m44.7s	98m47.9s		-k	0	3m24.3s	8m48.6s	52m11.8s
	-v	0	19m34.2s	27m44.3s	103m12.9s	nproc	*	0	1m40.4s	1m44.7s	47m22.6s
id	*	0	1m57.9s	1m59.7s	88m13.2s		–all	0	1m46.6s	2m22.1s	49m12.1s
	-g	0	1m52.1s	1m56.5s	79m13.6s	tac	*	0	1m50.9s	2m2.1s	47m40.6s
	-G	0	1m48.8s	1m58.7s	79m13.6s	uniq	*	0	1m25.5s	1m38s	50m57.4s
	-u	0	1m52.6s	1m59s	79m39.6s	stty	*	0	1m33.1s	1m49.7s	60m30.3s
hostid	*	0	1m14.3s	2m20.7s	78m16.4s	tty	*	0	1m36.6s	2m5.3s	46m19.3s

However, a difference is that they used the support of TPM for credentials. TEE has been recently applied to distributed computing, such as blockchain (e.g., [39]), making it possible to verify mathematically whether computing was correctly performed while maintaining security.

C. DEBUGGING

A characteristic of this research subject is that it only shows cases that become non-deterministic due to some other secondary factors in a single-threaded situation. As in [40]–[43], almost all attempts to make programs deterministic

TABLE 6. Full data set of coreutils evaluation.

Coreutils											
App	Opt	Err	O.T	C.T	D.T	App	Opt	Err	O.T	C.T	D.T
md5sum	*	0	2m15.8s	6m3.4s	68m43s	sync	*	0	9m41.4s	17m33.9s	45m29.3s
	-b	0	2m29.4s	6m5.6s	88m48s	uptime	*	0	1m42.5s	1m57.4s	67m47.4s
	-c	0	2m50.7s	6m1.5s	106m15.8s	tsort	*	0	1m3.6s	1m21.5s	43m49.8s
	-t	0	2m49.4s	6m1.4s	84m50.4s	uname	*	0	1m1.9s	1m13.8s	55m9.6s
who	*	0	3m55.1s	7m5.4s	76m1.1s		-s	0	1m1.8s	1m18.4s	45m28.4s
	-a	0	4m30.4s	7m9.7s	162m55.2s		-n	0	0m59.8s	1m18.5s	45m35.9s
	-b	0	3m54.4s	5m50.8s	122m20.9s		-r	0	1m8.8s	1m19s	46m9.8s
	-d	0	4m27.8s	6m0.9s	125m39.6s		-v	0	1m4.2s	1m14.8s	46m10.1s
	-H	0	4m47.3s	6m45.9s	126m32.9s	users	*	0	1m57.1s	2m11.7s	51m36.5s
	-l	0	4m22.9s	5m43.8s	81m50.7s	ptx	*	0	1m10.5s	1m29.1s	65m5.2s
	-m	0	4m42.2s	5m28.7s	108m14.4s		-A	0	1m10.8s	1m29s	66m57s
	-q	0	4m48.8s	5m23s	104m16.2s		-g	0	1m12.8s	1m24.1s	65m59.8s
	-r	0	4m53.1s	5m22.5s	115m44.5s	pinky	*	0	1m37.3s	1m45.2s	118m49.6s
	-s	0	5m34s	6m12.9s	118m33.4s		-l	0	1m10.8s	1m24.9s	71m51.6s

for effective debugging were on multi-threaded programs. A study [44] mentions multi-threaded execution as a major research subject, although there is similarity in that it deals with deterministic execution. Prior research adopted the solution of serializing the parallel execution flow, whereas this study differs in that it fixes changes caused by system calls or external factors to the program.

VIII. CONCLUSION

This study developed an execution hash generation tool using LLVM, verified the performance and occurrence of non-deterministic executions for binutils, coreutils, and OSS-Fuzz program sets, and discussed its feasibility as a PoW mechanism based on the findings. According to the test results, approximately 81 non-deterministic errors were observed among approximately 20 million executions using the tool for binutils and coreutils programs, and about 166 non-deterministic errors among approximately one billion executions for OSS-Fuzz programs. Certain cases were also selected among these non-deterministic errors to analyze and identify their causes. The developed tool is expected to be applicable to verifiable computing as well.

REFERENCES

[1] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive verifiable computing: Outsourcing computation to untrusted workers," in *Advances in Cryptology*. Berlin, Germany: Springer, 2010, pp. 465–482.

[2] S. Chen, J. H. Cheon, D. Kim, and D. Park. (2019). *Verifiable Computing for Approximate Computation*. Cryptology ePrint Archive. [Online]. Available: https://eprint.iacr.org/2019/762

[3] Z. Ghodsi, T. Gu, and S. Garg, "SafetyNets: Verifiable execution of deep neural networks on an untrusted cloud," in *Advances in Neural Information Processing Systems*, vol. 30. Red Hook, NY, USA: Curran Associates, 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/6048ff4e8cb07aa60b6777b6f7384d52-Paper.pdf

[4] X. Chen, "Introduction to secure outsourcing computation," *Synth. Lectures Inf. Secur., Privacy, Trust*, vol. 8, no. 2, pp. 1–93, Feb. 2016.

[5] X. Yu, Z. Yan, and A. V. Vasilakos, "A survey of verifiable computation," *Mobile Netw. Appl.*, vol. 22, no. 3, pp. 438–453, May 2017, doi: 10.1007/s11036-017-0872-3.

[6] D. Fiore, R. Gennaro, and V. Pastro, "Efficiently verifiable computation on encrypted data," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2014, pp. 844–855.

[7] X. Yu, Z. Yan, and R. Zhang, "Verifiable outsourced computation over encrypted data," *Inf. Sci.*, vol. 479, pp. 372–385, Apr. 2019.

[8] C. Lattner. (2008). *LLVM and Clang: Next Generation Compiler Technology LLVM: Low Level Virtual Machine*. Accessed: Jun. 31, 2021. [Online]. Available: https://reup.dmcs.pl/wiki/images/0/09/53_BSDCan2008ChrisLattnerBSDCompiler.pdf

[9] T. Liu, C. Curtsinger, and E. D. Berger, "Dthreads: Efficient deterministic multithreading," in *Proc. 23rd ACM Symp. Operating Syst. Princ. (SOSP)*, 2011, pp. 327–336.

[10] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data race fuzzing for kernel file systems," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 1643–1660, doi: 10.1109/SP40000.2020.00078.

[11] C. Lemieux and K. Sen, "FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, Sep. 2018, pp. 475–485.

[12] 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.

[13] R. Shakya, J. Gibson, and J. Brackins, "Fuzzing to identify undiscovered bugs in scientific software," *Proc. Student Res. Creative Inquiry Day*, vol. 4, no. 1, May 2020. [Online]. Available: https://publish.tntech.edu/index.php/PSRCI/article/view/679

[14] GNU Org. *Binutils—GNU Project—Free Software Foundation*. Accessed: Jun. 2, 2021. [Online]. Available: https://www.gnu.org/software/binutils/

[15] GNU Org. *Coreutils—GNU Core Utilities*. Accessed: Jun. 2, 2021. [Online]. Available: https://www.gnu.org/software/coreutils/

[16] G. License, "Gcov: Gnu coverage tool," Tech. Rep. [Online]. Available: http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_8.html

[17] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *Proc. 14th USENIX Workshop Offensive Technol. (WOOT)*, 2020.

[18] J. C. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," *ACM Trans. Comput. Syst.*, vol. 15, no. 3, pp. 217–252, Aug. 1997.

[19] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "PIN: A binary instrumentation tool for computer architecture research and education," in *Proc. Workshop Comput. Archit. Educ. Held Conjunct 31st Int. Symp. Comput. Archit. (WCAE)*, 2004, p. 22.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2005, pp. 190–200.

[21] K. Serebryany, "Libfuzzer—A library for coverage-guided fuzz testing," LLVM Project, Tech. Rep., 2015.

[22] V. Gligor and M. Woo, "Requirements for root of trust establishment," in *Security Protocols*. Cham, Switzerland: Springer, 2018, pp. 192–202.

[23] W. Li, Y. Xia, and H. Chen, "Research on ARM TrustZone," *GetMobile, Mobile Comput. Commun.*, vol. 22, no. 3, pp. 17–22, Jan. 2019.

[24] V. Costan and S. Devadas. (2016). *Intel SGX Explained*. ePrint IACR. [Online]. Available: https://eprint.iacr.org/2016/086

[25] G. Kumar, R. Saha, M. Rai, R. Thomas, and T. H. Kim, "Proof-of-work consensus approach in blockchain technology for cloud and fog computing using maximization-factorization statistics," *IEEE Internet Things J.*, vol. 6, no. 4, pp. 6835–6842, Aug. 2019.
[26] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 3–16.
[27] D. Firesmith. *The Challenges of Testing in a Non-Deterministic World*. SEI Blog. Accessed: May 1, 2022. [Online]. Available: https://insights.sei.cmu.edu/blog/the-challenges-of-testing-in-a-non-deterministic-world/
[28] S. Mathews. (Feb. 23, 2021). *How to Avoid the Headache With Non-deterministic Bugs in Software*. Accessed: May 1, 2022. [Online]. Available: https://levelup.gitconnected.com/how-to-avoid-the-headache-with-non-deterministic-bugs-in-software-e24457f05c9b
[29] M. Ahmadvand, A. Hayrapetyan, S. Banescu, and A. Pretschner, "Practical integrity protection with oblivious hashing," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, Dec. 2018, pp. 40–52.
[30] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. H. Jakubowski, "Oblivious hashing: A stealthy software integrity verification primitive," in *Information Hiding*. Berlin, Germany: Springer, 2003, pp. 400–414.
[31] *CS-TR-06-1: Program Trace Formats for Software Visualisation*. Accessed: Jun. 30, 2021. [Online]. Available: http://www.mcs.vuw.ac.nz/comp/Publications/CS-TR-06-1.abs.html
[32] P. Golle and I. Mironov, "Uncheatable distributed computations," in *Topics in Cryptology*. Berlin, Germany: Springer, 2001, pp. 425–440.
[33] W. Du, J. Jia, M. Mangal, and M. Murugesan, "Uncheatable grid computing," in *Proc. 24th Int. Conf. Distrib. Comput. Syst.*, 2004, pp. 4–11.
[34] P. Golle and S. Stubblebine, "Secure distributed computing in a commercial environment," in *Financial Cryptography*. Berlin, Germany: Springer, 2002, pp. 289–304.
[35] T. W. Lockhart, "The design of a verifiable operating system kernel," Tech. Rep., Univ. Brit. Columbia, Endowment Lands, BC, Canada, 1979.
[36] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not," in *Proc. IEEE Trustcom/BigDataSE/ISPA*, Aug. 2015, pp. 57–64.
[37] Z. Shen and X. Wu, "The protection for private keys in distributed computing system enabled by trusted computing platform," in *Proc. Int. Conf. Comput. Design Appl.*, Jun. 2010, p. 576.
[38] M. Achemlal, S. Gharout, and C. Gaber, "Trusted platform module as an enabler for security in cloud computing," in *Proc. Conf. Netw. Inf. Syst. Secur.*, May 2011, pp. 1–6.
[39] L. P. Maddali, M. S. D. Thakur, R. Vigneswaran, M. A. Rajan, S. Kanchanapalli, and B. Das, "VeriBlock: A novel blockchain framework based on verifiable computing and trusted execution environment," in *Proc. Int. Conf. Commun. Syst. Netw. (COMSNETS)*, Jan. 2020, pp. 1–6.
[40] J. Trümper, J. Bohnet, and J. Döllner, "Understanding complex multi-threaded software systems by using trace visualization," in *Proc. 5th Int. Symp. Softw. Visualizat. (SOFTVIS)*, 2010, pp. 133–142.
[41] S. Taheri, I. Briggs, M. Burtscher, and G. Gopalakrishnan, "DiffTrace: Efficient whole-program trace analysis and diffing for debugging," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, Sep. 2019, pp. 1–12.
[42] J. Burnim and K. Sen, "Asserting and checking determinism for multi-threaded programs," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng. Eur. Softw. Eng. Conf. Found. Softw. Eng. Symp. (ESEC/FSE)*, 2009, pp. 3–12.
[43] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang, and Y. Liu, "MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs," in *Proc. 29th USENIX Secur. Symp. (USENIX Security)*, 2020, pp. 2325–2342.
[44] C. Zamfir and G. Candea, "Execution synthesis: A technique for automated software debugging," in *Proc. 5th Eur. Conf. Comput. Syst. (EuroSys)*, 2010, pp. 321–334.

EUNYEONG AHN is currently a Researcher in convergence security engineering with Sungshin Women's University, Seoul, South Korea. Her research interests include system security, fuzzing, compiler/LLVM, hacking, and autonomous car security.

SUNJIN KIM received the B.S. degree in convergence security engineering from Sungshin Women's University, South Korea, in 2021, where she is currently pursuing the M.S. degree in future convergence technology engineering. Her research interests include system security, computer vision, and attacking machine learning with adversarial attack.

SAEROM PARK received the B.S. and Ph.D. degrees in industrial engineering from Seoul National University, in 2013 and 2018, respectively. She is currently an Assistant Professor with the Department of Convergence Security Engineering, Sungshin Women's University, Seoul, South Korea. Her research interests include secure and stable machine learning, stability analysis, robust training from an adversary, and privacy-preserving machine learning through encryption.

JONG-UK HOU (Member, IEEE) received the B.S. degree in information and computer engineering from Ajou University, South Korea, in 2012, and the M.S. and Ph.D. degrees from KAIST, South Korea, in 2014, and 2018, respectively. He has been an Assistant Professor with the School of Software, Hallym University, since 2019, and the Principal Investigator of the Multimedia Computing Laboratory. His research interests include various aspects of information hiding, point cloud processing, computer vision, machine learning, and multimedia signal processing.

DAEHEE JANG received the Ph.D. degree in information security from KAIST, in 2019. He is currently an Assistant Professor with the Security Engineering Department, Sungshin Women's University. He worked as a Postdoctoral Researcher at Georgia Tech, until 2020. He participated in various global hacking competitions (such as DEFCON CTF) and won several awards. He received a special prize from 2016 KISA annual event for finding zero day security vulnerabilities in many software products. He is also the Founder of pwnable.kr wargame—an education platform for training hacking skills.