Received March 28, 2017, accepted May 15, 2017, date of publication May 19, 2017, date of current version June 28, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2706943

A Branch History Directed Heuristic Search for Effective Binary Level Dynamic Symbolic Execution

YAN HU[1], WEIQIANG KONG[1], YIZHI REN[2], AND KIM-KWANG RAYMOND CHOO[3], (Senior Member, IEEE)
[1]School of Software, Dalian University of Technology, Dalian 116023, China
[2]School of Cyberspace, Hangzhou Dianzi University, Hangzhou 310018, China
[3]Department of Information Systems and Cyber Security, The University of Texas at San Antonio, San Antonio, TX 78249-0631, USA

Corresponding author: Kim-Kwang Raymond Choo (raymond.choo@fulbrightmail.org)

This work was supported by the National Natural Science Foundation of China under Grant 61300017 and Grant 61572097.

ABSTRACT Heuristic search is an important part of modern dynamic symbolic execution (DSE) tools, as heuristic search can be used to effectively explore the large program input space. Searching task remains one of several research challenges due to the fact that the input space grows exponentially with the increase of program size, and different programs may have very different structures. The challenge is compounded in a cyber-physical system or cloud-based Internet of Things environment. In this paper, we propose a novel heuristic search algorithm, which analyzes the program execution history and uses the refined history information to inform the search. This paper is based on the observation that the branch and input history generated during dynamic symbolic execution can help memorize the explored input space, and infer the partial structure of the program. With a summarized branch history, the proposed heuristic search makes informed (and better) decisions about which input area to search next for better efficiency. To evaluate the search algorithm, we implement the core DSE engine, integrated with modules to perform execution history collection and analysis. To make our method practical, we incorporate taint analysis and constraint solving statistics to guide the search algorithm. Experimental results demonstrate that with the rich history information, the new search algorithm can explore the input space more effectively, thus resulting in detecting software defects faster.

INDEX TERMS Dynamic symbolic execution, branch history, test case generation, heuristic search, cyber-physical system.

I. INTRODUCTION

Automated test case generation is an important technique in software testing. As software systems become increasingly complex (e.g. due to the need for more functionality and features), it is challenging in practice to manually and comprehensively test the software. Testing automation is one solution to dealing with the challenges in testing of large complex software, such as those deployed in cyber-physical systems and cloud-based Internet of Things environment (also referred to as Cloud of Things [1], [2]). Random testing and structural testing are two typical ways of automated testing, which have been extensively studied by the software engineering community.

Random testing [3]–[5] has been used in automatic unit test generation, and proven to be rather effective in generating unit test cases. In random testing, random inputs are generated and given as input to the program under test in order to achieve good coverage. A limitation of random testing is that it has no knowledge of the program's inner structure, and may not be able to generate high coverage test cases.

Structural testing differs from random testing. As program structures are important for testing [6] and security analysis [7], [8], program structures are taken into consideration during the test case generation process in structural testing. The program control flow or data flow is observed, and used to guide the generation of test inputs to explore different program paths. Dynamic symbolic execution (DSE) is a typical structural testing technique. A DSE procedure steers program execution path by manipulating and solving symbolic path constraints. Therefore, DSE tools are able to understand the program structure, and perform better structural testing. The studies in [6] and [9] demonstrated that DSE can be rather effective, and scale to very large software code base.

Software testing using dynamic symbolic execution has been extensively explored in the literature [9]–[15]. For example, the approach in [9] works by transforming source code into bitcode [16], and performing dynamic symbolic execution on the bitcode. CREST [17] relies on source code instrumentation to retrieve branch conditions, and then manipulates the path conditions to generate new inputs. Other than DSE with source code transformation, researchers have also attempted to apply dynamic symbolic execution directly on binary code. By working directly at the binary level, the complexity of manipulating source code is avoided. Binary level analysis can be especially useful for security analysis [18]. SAGE [12] and EXE [13] are two well known binary level symbolic executors. Although binary level dynamic symbolic execution tools are currently very popular in software testing area, few of them are publicly available, with Fuzzgrind [19] as an exception. Fuzzgrind is an open source tool, based on Valgrind [20], the widely known Dynamic Binary Instrumentation (DBI) tool. It is based on a dynamic symbolic execution engine, and is able to generate exploits for bugs in real-life applications. A depth-first search algorithm is implemented in Fuzzgrind, and used in the process of input space exploration.

During our preliminary evaluation of the original Fuzzgrind, we determined that it suffers from repeated constraint solving, and duplicate inputs; thus, rendering Fuzzgrind inefficient. In this work, we build our own Valgrind-based dynamic symbolic executor (hereafter referred to as DigEXE – Directed Input Generation with EXEcution feedback). The enhancement we made in DigEXE is an execution feedback component, which manages the execution history collected during the symbolic execution process. Based on DigEXE, we propose our new branch-history guided search heuristics. Specifically, we regard the contributions of this paper to be as follows:

1) We propose a branch history guided heuristic search. We maintain the input pool and branch history during the symbolic execution process. With the execution history, we can prevent duplicate constraint solving, and filter out duplicate inputs. The history information is also used to prioritize the inputs in the input pool, deciding which input is the best to be selected as seed in the next round of input generation.
2) We design the DigEXE symbolic execution framework, and integrate the new heuristic search algorithm. As DigEXE is based on dynamic binary instrumentation technique, it can perform symbolic execution directly on compiled binary programs.

We then evaluate the branch history directed heuristic search by analyzing real-world software projects with our DigEXE implementation. The experimental results demonstrate the effectiveness of the proposed search algorithm.

In the next section, we describe the preliminaries necessary to understand the remainder of this paper.

II. PRELIMINARIES

Program execution history is regarded as a source of feedback in this paper. It can be carefully analyzed, and used to guide the search process in automated test input generation. In this section, we define the relevant notations, and explain the concept of ‘‘execution feedback’’.

We start from the most direct source of execution feedback, namely: program trace (Definition 1). Each program run generates a program trace, which is a sequence of operations. The sequence of operations represents the program execution under a certain input.

Definition 1 (Program Trace): A program trace is defined as $\tau =< o_1, o_2, \ldots, o_n >$, where o_i is a program operation, $(1 \leq i \leq n)$.

A program trace is generated after each monitored run of the test program. Definition 1 gives a generic definition of program trace. In reality, program traces could be at different abstract levels. At the finest granularity, a program trace can contain details of every instruction level operation. A program trace can also be as simple as an API call sequence.

Program traces are produced to help understand the runtime behavior of a program under test. Dynamic program analyzers rely on program traces to construct a concrete vision of the program behavior. Different program traces are used in different analyzers. Light-weight analyzers use program traces at a higher abstract level, which is not very precise but is less costly. Analyzers of high precision use low level program trace, which incurs additional cost in order to obtain more detailed and precise program trace.

In practice, monitor codes are inserted into the program to monitor program behavior, and generate program traces. They are actually some control code at interesting program locations. The insertion of monitor code is normally done by program instrumentation, which is a common practice in program analysis and software testing. The form of monitor code is decided by the purpose of the analysis. For example, if we wish to perform a basic block coverage analysis, monitoring code should be inserted at the basic blocks. If we wish to perform deeper analysis, we may want to insert additional code at each instruction, and monitor all the execution details of each instruction.

If the analyzer requires analysis of detailed memory operations in the test program, then we should add monitor code at memory access locations. During program execution, the monitor code collects information about the address of the accessed memory, the type of memory access (read or write), and the location of the memory access in the program code. The memory monitor can then log memory accesses, and perform further analysis on the collected memory accessing data as a whole. Even a small program may have many memory accesses. Therefore, it is very costly to observe every memory access during program execution. Hence, data flow analysis should be applied so that memory tracers could record only those memory locations that are actually influenced by program input. This is presented in more detail in Section V.

In order to identify each element in a program trace, we explicitly define another notation: Program Location (see Definition 2).

Definition 2 (Program Location): A program location is a position in the test program.

A program location can be a tag to distinguish one program operation from another. If source level information can be obtained by the analyzer, then we may use line number in a source file to mark a program location. When the analysis is performed on program binaries without debugging information, a specific program location can be identified by the corresponding stack trace. Information of program location is an important part of the history information, which will be used to select appropriate path conditions to invert during the dynamic symbolic execution process.

Program input(Definition 3) must be supplied to run the test program.

Definition 3 (Program Input): A program input tuple is $input = < byteArray, state, sig, parent, depth >$, where $byteArray$ is the actual input data, each input has a $state$, and a signature sig.

In practice, input data can have different forms, like strings from command line, environment variables, or binary stream from a file. For simplicity, we take the program input as a finite length sequence of byte data $byteArray$, stored in an input file. Each input is given a $state$ to indicate its different usage during our history guided dynamic symbolic execution process (to be explained in subsection III-B). After program execution with the given input, a valid signature (see Definition 4) will be generated and assigned to the input structure as sig. If the input has a valid $parent$ value, then it means that the $input$ is generated by inverting the path of $parent$ input at $depth$.

Definition 4 (Input Signature): An input signature is a path prefix, $Psig =< B, V, size >$, where B is the vector of branches in the path, V is the vector of Boolean values indicating whether true or false branch is taken. $size$ is an integer representing the length of the input signature.

In our DigEXE tool, the input signature is used as a tag for a specific program input. The signature is constructed from the program trace after the program has been executed with a given input. DigEXE use the input signatures to prioritize and filter inputs in the input pool.

In the next section, we describe our proposed approach.

III. OUR PROPOSED APPROACH

The focus of our work in this paper is to design a method to effectively monitor the dynamic symbolic execution process, collect and process program traces into specific form of feedback information, which is then used by the new search algorithm to guide the search process.

Our approach is based on dynamic symbolic execution , which generally contains three major steps, namely:

1) Prepare the initial input, and place the new input in the list of input candidates.
2) Select one input from the candidates, execute it. If a bug manifests in the execution or the stop criteria of the search algorithm is met, then exit; otherwise, go to the next step.
3) For each path depth, invert the corresponding branch condition, solve the newly generated path constraints, generate a new input and place it into the queue of input candidates. Return to step 2.

We implement the dynamic symbolic execution engine in our DigEXE framework. The search algorithm is integrated with the input selection in step 2.

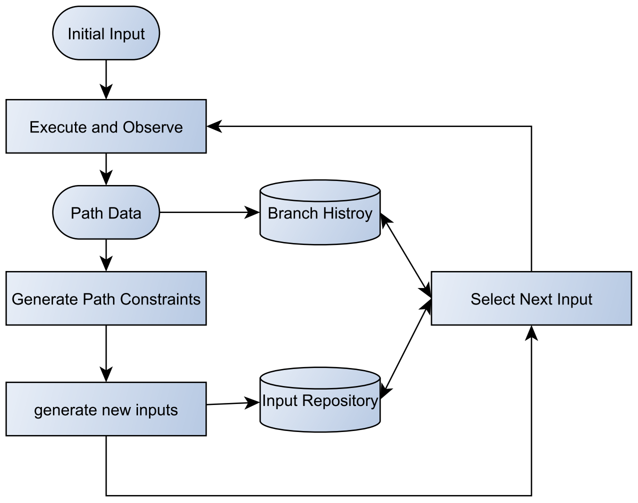


FIGURE 1. Workflow of DigEXE.

A. THE WORKFLOW OF DigEXE

The workflow of the branch history directed dynamic symbolic execution process is illustrated in Fig. 1.

In the "Execute and Observe" component, an execution monitor observes each program run, and records the program trace with configured granularity. We use several monitors to generate different types of program traces to meet the requirements of different analysis purposes. The execution monitor can be injected into the test program with static or dynamic instrumentation technique. Since DigEXE is designed to work on unmodified binaries, we use dynamic instrumentation to inject the monitor code at appropriate program locations. The main advantage of dynamic instrumentation is that we can do unobtrusive analysis of the program, and focus on the designing of search strategies to explore the input space of test programs.

A trace analyzer will process the generated program traces, mainly to collect the conditional path information. We then update the branch history with the information of each branch in the recorded program path.

The "Generate Path Constraints" component is responsible for producing path constraints from the collected program trace. Path constraints are processed into the form of constraint solver queries. Next in the ''Generate new inputs'' phase, the constraint query is used as a blueprint to generate new queries that can lead to new inputs. The constraint query is mutated at specific path depth, following the dynamic symbolic execution principle.

New inputs generated are put into the candidate input queue. The ''Select Next Input'' phase will select one input from those candidates, execute it, and search its neighborhood for more new inputs. The expanded inputs are usually discarded in a standard dynamic symbolic execution implementation. In DigEXE, we still record the input data, and its path signature, to help filter out duplicate inputs in the input history. The expanded inputs information is also used to avoid repeated flipping of a branching condition at the same input depth of similar inputs.

The heuristic search algorithms play an important role in the dynamic symbolic execution process. Starting from an initial input, each input is executed and expanded, generating several NEW inputs accordingly. Those NEW inputs are called children of the EXPANDED input. The number of child inputs depends on the length of execution path.

The inputs naturally form a tree structure. For real world applications, the input tree can be very large. Exploring the complete program space represented by the input tree is impractical. However, heuristic search algorithms can help us explore the most valuable / useful subspaces.

B. MAINTENANCE OF EXECUTION HISTORY

The search algorithm proposed in this paper uses the execution history to guide its search procedure. It keeps track of two types of feedback information, namely: branch history, and input repository.

The input repository remembers all the different inputs generated. The execution of each DigEXE phase can affect the state of a program input. Throughout the DigEXE execution loop, each program input may undergo several state changes. The input repository is responsible for maintaining the life cycle of program inputs, tracking and updating their states.

In DigEXE, program inputs may be in any of the following states, namely: (1) NEW; (2) COV-ANALYZED; (3) EXECUTED; (4) EXPANDED.

A program input is in the NEW state, when it has just been constructed. In the NEW input, input variables are assigned with meaningful initial values. The initial values could be generated randomly (for the initial input), or obtained from the solutions of the constraint solver.

DigEXE uses a light-weight profiler to generate coverage data. With light-weight coverage monitors, the test program executes with a given program input, and generates coverage information. After that, the state of the program input changes from NEW to COV-GENERATED. The state transfer process is shown in Fig. 2. The coverage information is used as one factor when selecting the next input to be expanded.

We rely on a heavy-weight execution monitor to generate detailed memory traces. After the heavy-weight analysis,

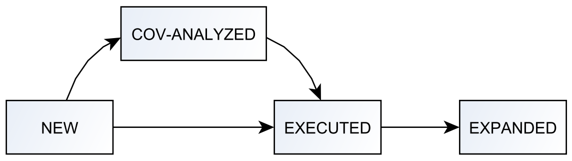


FIGURE 2. Input state transfer diagram.

a program trace is generated. We then update the state of the input to EXECUTED. The detailed program trace contains all program operations of interest, including memory accesses.

The state of the input transfer from EXECUTED to EXPANDED, after the DSE procedure has tried every possible branch inversion. With the detailed program trace associated with an EXECUTED input, DigEXE can then flip one of the path conditions, build a query for constraint solver, and solve the query to generate new inputs. After our DSE tool tried all possible flippings of path conditions, the state of the input is changed to EXPANDED.

Based on the input repository, we also calculate important accumulative information, namely: (1) branch statistics; (2) program location statistics. We extract branch information from an EXECUTED input, and update the branch coverage count, branch flipping counts. Those information is kept in a separate branch history.

IV. DESIGN OF SEARCH STRATEGIES

In this section, we describe the search strategies in our DigEXE framework presented in the preceding section, and explain how we use program execution history to guide the search process. We revisit the baseline search algorithm (DFS algorithm), before presenting our new history directed search algorithm.

A. DFS ALGORITHM

The DFS algorithm (see Algorithm 1) is the same as the one implemented in Fuzzgrind [19].

In Algorithm 1, an initial input is first created with a random input seed file (line 1). A memory tracer is then created (line 2) to generate detailed program traces (see Definition 1) required by the dynamic symbolic execution. The memory tracer is executed to generate a program trace for the selected input (line 7). It is a Valgrind plugin that follows the program execution path, recording memory accesses and path conditions. The trace is then expanded in the for loop (lines 8-12), following the standard dynamic symbolic execution procedure. The invertAndSolve procedure(line 9) is the core part, which inverts a branch condition at certain depth in the path and solves the resulting new path condition to generate new inputs. After completing the expansion iteration of one input, several new inputs are generated and inserted into the input queue. The updated input queue is then sorted in decreasing order of the invert depth of each input's parent input (line 13), and the new input at the head of input queue is chosen as the next candidate for input expansion (line 6).

The Fuzzgrind tool implements Algorithm 1 as the main search algorithm. This DFS algorithm may create duplicates

Algorithm 1 DFS Search Algorithm

```
   Input: initialFile = the input seed file for the input
          search process
1  input = new Input(intialFile) ;
2  memTracer = new MemoryTracer();
3  inputQueue.append(input);
4  loopIter = 0;
5  while not allBranchCovered do
6      input = inputQueue.popFront();
7      memTracer.exec(input);
8      for depth = 0; depth $\leq$ pathLen(input) do
9          newInput = invertAndSolve(input, depth);
10         if newInput != NULL then
11             inputQueue.append(newInput);
12             write the content of newInput into input file;
13     sortByDepth(inputQueue);
14     loopIter = loopIter + 1;
15     if loopIter < MAX_LOOP then
16         break;
```

Algorithm 2 DFS-i Search Algorithm

```
1  input = new Input(intialFile) ;
2  memTracer = new MemoryTracer();
3  newInputQueue.append(input);
4  expandedInputQueue = { };
5  loopIter = 0;
6  while not allBranchCovered do
7      input = newInputQueue.popFront();
8      memTracer.exec(input);
9      path = memTrace.getProgramPath(input);
10     for depth = 0; depth $\leq$ len(path) do
11         expandCandidate = invert(path, depth);
12         isExpanded = false;
13         for eInput in expandedInputQueue do
14             if match(input, eInput, depth) then
15                 isExpanded = true;
16         if !isExpanded then
17             newInput = solve(input, depth);
18             if newInput == NULL then
19                 continue;
20             newInput.parent = input;
21             newInput.invertDepth = depth;
22             newInputQueue.append(newInput);
23     sortByDepth(inputQueue);
24     if !expandedInputQueue.hasMatch(input.sig) then
25         expandedInputQueue.append(input);
26     loopIter = loopIter + 1;
27     if loopIter < MAX_LOOP then
28         break;
```

in the input queue, which relies on the input signature to filter out duplicate inputs. In this case, we first attempt to improve the DFS search process with the input repository as the execution feedback. We name the enhanced DFS search algorithm DFS-i (The "i" means "input").

The DFS-i algorithm is described in Algorithm 2. DFS-i keeps all the expanded inputs in a separate queue. Before an input is placed in the queue of the expanded inputs, DFS-i checks its input signature against those already in the queue, and only adds it to the expanded queue when there is no input with the same signature in the queue.

In DFS-i, when we attempt to invert an input (line 11), we first check that the inversion has not been tried before, by calling match(input, eInput, depth) for each eInput in the expanded queue (lines 13-15). The match procedure checks if input.sig and eInput.parent.sig have the same prefix with length *depth*. In this way, DFS-i avoid repeated solving of the same inverted path condition. When an input has been fully expanded, and if its signature is different from all the signatures of inputs in expandedInputQueue, then we add it to the expandedInputQueue (lines 24-25).

Considering the performance of DSE tool, we use fast but not very accurate taint analysis. In this case, the path condition supplied to the DSE tool may be incomplete, and the DFS algorithm may generate duplicate inputs. As the DFS-i algorithm eliminates the possibility of duplicate inputs, it is clear that the DFS-i algorithm is more efficient than the DFS algorithm.

B. BHHS - BRANCH HISTORY BASED HEURISTIC SEARCH

We now present the new heuristic search BHHS (see Algorithm 3), which uses both input repository and branch history as the guidance information.

In our BHHS algorithm, we establish the mapping between program inputs and branch information. The map biMap is initialized at line 6. A memory tracer is still required to monitor the program execution and generate detailed program traces containing memory accesses. The tracer *MemTracer* is initialized at line 7 in BHHS.

The executed input is expanded by trying to invert branch condition at each depth in the program path. Each newly generated input is going to be added to the new input queue, waiting to be selected for execution and expansion.

Before actually solving the inverted path, the expandCandidate is checked against the expandedInputQueue (lines 16-18). If the expandCandidate (line 14) does not match any signature of any expanded inputs in expandedInputQueue, then we start solving the inverted query (line 20). If the constraint solving is not successful, then the variable newInput is set to NULL. This case is handled in lines 21-22. If the constraint solving successfully emits a valid newInput, then we associate the newInput with the inverted branch by adding the newInput into the biMap (lines 29-31), and update the branch inversion count (line 31). After the current input

Algorithm 3 BHHS Heuristic Search

```
1  input = new Input(intialFile);
2  covTracer = new CoverageTracer();
3  covTracer.exec (input);
4  covTracer.updateBranchHistory();
5  input.state = COV-ANALYZED;
6  biMap = new BranchInputMap();
7  memTracer = new MemoryTracer();
8  expandedQueue = { };
9  while true do
10     memTracer.exec(input);
11     input.state = EXECUTED;
12     path = memTrace.getProgramPath(input);
13     for depth = 0; depth ≤ len(path) do
14         expandCandidate = invert(path, depth);
15         isExpanded = false;
16         for eInput in expandedInputQueue do
17             if match(input, eInput, depth) then
18                 isExpanded = true;
19         if !isExpanded then
20             newInput = solve(input, depth);
21             if newInput == NULL then
22                 continue;
23             covTracer.exec (newInput);
24             covTracer.updateBranchHistory();
25             newInput.state = COV-ANALYZED;
26             newInput.parent = input;
27             newInput.parentInvertDepth = depth;
28             lastInvertedBranch = input.path[depth];
29             inputs =
               biMap[lastInvertedBranch].invertedInputs;
30             inputs.insert(newInput);
31             biMap[lastInvertedBranch].invertedTimes++;
32     expandedInputQueue.insert(input);
33     if stopCriteriaMet() == true then
34         break;
35     branch = getBestBranch(biMap);
36     newInputs = biMap[branch].invertedInputs;
37     sortedInputs = sortByInvertDepth(newInputs);
38     input = getFirst(sortedInputs);
39     biMap[branch].remove(input);
```

is fully expanded, it is inserted into expandedInputQueue (line 32).

When a NEW input is created, BHHS uses a light-weight coverage analyzer *covTracer* to analyze the branch coverage information, and incrementally updates the branch history (*covTracer*.updateBranchHistory(), as in line 4 and line 24).

When BHHS starts the next input selection, it first chooses the most appropriate branch by its own standard (line 35). In the current implementation, we choose the newly covered branch as the best branch candidate. If there is no newly covered branche, then we choose the branch with the least inversion count (minimal biMap[branch].invertedTimes). With the selected branch, BHHS then choose one input with the largest invert depth as the next input to be executed and expanded (lines 36-39).

V. IMPLEMENTATION

In this section, we briefly describe the implementation of BHHS algorithm on our DigEXE framework.

A. DYNAMIC BINARY INSTRUMENTATION

The execution monitor of DigEXE is implemented as the plugin of Valgrind [20]. Valgrind is a popular dynamic binary instrumentation (DBI) tool. DBI tools are widely used in program analysis, software profiling, testing, and computer security research. Valgrind, Dynamorio [21] and PIN [22] are three most popular DBI tools, which act as the basic program analysis platform. With their plugin system, it is easy to build customized dynamic binary analysis tools such as program property checkers or profilers.

We implement the light-weight coverage analysis plugin to collect coverage statistics. The coverage monitoring checks only at the basic block level, with less runtime overhead than the heavy-weight memory access monitor.

As DigEXE is built upon Valgrind, it is able to analyze binary program without the need for modification.

B. SYMBOLIC EXECUTION

Our work in this paper is based on symbolic execution [11], which is an effective program analysis technique. It is widely used in the area of program verification, and software testing. Static symbolic execution is widely used in verification of program properties, while dynamic symbolic execution is more suited for software testing.

The input generation procedure of our DigEXE tool uses dynamic symbolic execution. It uses STP constraint solver [23] to solve path constraints. A heavy-weight memory access monitor captures detailed program traces, and constructs STP solver queries.

Path explosion is the main challenge of applying symbolic execution to real world software. When program size increases, we will see an exponential increase in program paths. Therefore, generating inputs to cover all those paths is impossible. Heuristic search is integrated into dynamic symbolic symbolic execution tools to alleviate the problem. In DigEXE, We implemented search algorithms, trying to help generate high quality test cases within reasonable time.

C. TAINT ANALYSIS IN DigEXE

We can track the whole data flow of the test program, and obtain a precise view of the program behavior. However, if we do so, we will have to use heavy-weight execution observer, whose execution is time consuming. In this case, a very lengthy program trace will be generated, and require more time to process the trace and retrieve useful information. In practice, it is unnecessary to record detailed behavior of each instruction.

When exploring the input space using the DSE technique, we are more interested in memory variables that depend on the input data which may affect branching conditions. Taint analysis is a good choice to handle such situation.

Taint analysis [24], [25] is a dataflow analysis technique that is commonly used in computer security research. The main purpose of taint analysis is to detect malicious information flow that can cause the leakage of confidential data in a security-sensitive program.

Taint analysis tool tracks the def-use chain initiated by tainted variables. It can be considered a testing method, which observes and checks the program information flow rather than verifying the non-interference property. Taint analysis is somewhat similar to program dependence analysis in program slicing.

In the implementation of DigEXE, we use taint analysis to reduce the program traces, and speed up the DSE procedure. Initially, we track and taint all the input data. Then, the taint analysis is applied to help us focus on the propagation of tainted input data in the program flow. Only operations on tainted memories and tainted branch statements are recorded in the program trace. Thus, program elements that are not affected by the input data will not be recorded in the trace.

Taint analysis in DigEXE is integrated with the heavy weight execution monitor responsible for generating program traces containing detailed memory accesses. For performance reasons, we consider taint propagation on dataflow, and ignore control flow taint propagation.

DigEXE mainly uses a input file as the program input. Its taint tracking process relies on the observation of file open and read system calls. The content of the file is read into a memory buffer, which then acts as the tainted source. Input data is then propagated via load/store operations, assignment and other operations that can cause dataflow.

We maintain a tainted set of memory addresses and registers. For each binary operation, we update the tainted set accordingly. Therefore, after execution of the last statement in the basic block, we obtain a new map of tainted variables. The tainted information at the basic block exit is used to decide whether the following branching statement is tainted (or not). We then track only branches dependent on tainted memory or registers.

VI. EVALUATION

We implemented the branch history directed heuristic search algorithm in our DigEXE tool, which is based on the Valgrind dynamic instrumentation framework. The flexible plugin system of Valgrind makes it easy to develop and integrate various profilers and analyzers together, which is a basic requirement to drive the history directed input generation process. Tracers of different granularity are implemented to conduct tasks from light-weight profiling to heavy-weight path analysis. Tracers and branch history maintainer work together with the constraint solving component to generate inputs. The constraint solver used in our work is the STP solver [23].

TABLE 1. Description of selected benchmarks.

benchmark	Description
cjpeg	jpeg processing library and utilities
convert	image conversion
js	Mozilla JavaScript interpreter
ld	GNU program loader
qtdump	movie dumper
readelf	read elf file format, from GNU binutils
tiffinfo	tiff format parser

A. EXPERIMENT SETUP

All experiments were conducted using a 32bit Ubuntu Linux 10.04 system (Intel Core Duo T7100, 4G memory) on real world open source projects - see TABLE 1.

As previously discussed, DFS-i is the improved version of the standard DFS algorithm implemented in Fuzzgrind. As DFS-i prunes already attempted inputs from been expanded and executed, we can be assured that DFS-i outperforms DFS. We now compared between both BHHS and DFS-i algorithms.

B. COVERAGE STATISTICS

Branch coverage is the main criteria for evaluating BHHS performance. The comparative summary of the BHHS heuristic and DFS-i heuristic is given in Fig. 3, which demonstrates that BHHS outperforms DFS-i and achieves better coverage with the given 50 execute-and-expand iterations.

We also checked the pool of generated inputs, and found that BHHS can steadily generate more variant new inputs that DFS-i. This is mainly because BHHS prioritizes the search towards most recently covered new branches. For details on the input growth, we refer the reader to Fig. 4.

C. CAPABILITY OF BUG DETECTION

We also compared the bug detection capabilities of the two search algorithms implemented in our DigEXE framework. DigEXE with our BHHS heuristic successfully generated a few error inducing inputs with the 50 execute-and-expand iterations. It shows that BHHS heuristic also outperfoms DFS-i in terms of error detection capability (see TABLE 2).

For benchmarks such as convert, js, ld, and readelf, the input file formats are complex and carry extra semantic information. In the future, we will integrate grammar directed input generation techniques [26] to improve DigEXE error detection capability.

VII. RELATED LITERATURE

Heuristic search has wide applications in machine learning [27], [28], image processing [29], etc.. In this paper, we explore the application of search algorithms to the task of automatic test input generation.

RANDOOP [30] is a unit testing framework for Java programs. This is an example of a random search algorithm used

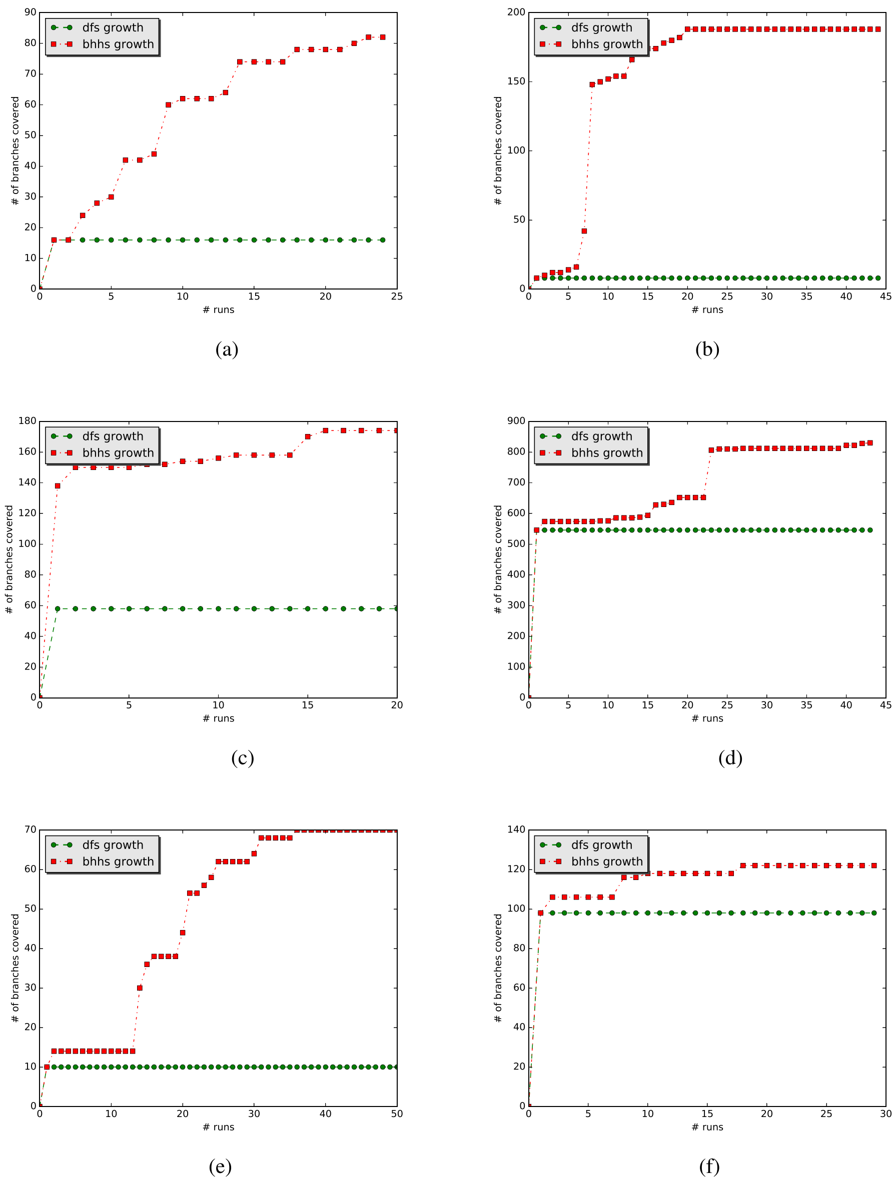


FIGURE 3. Branch growth satistics of the benchmark programs. (a) cjpeg. (b) convert. (c) js. (d) ld. (e) readelf. (f) tiffinfo.

TABLE 2. The number of error inducing inputs generated.

benchmark	DFS-i	BHHS
cjpeg	0	2
convert	0	0
js	0	0
ld	0	0
qtdump	0	3
readelf	0	0
tiffinfo	0	0

in unit test generation, and has been shown to be capable of generating high coverage unit test cases.

CREST [17] integrates different search algorithms, and the authors conducted an empirical study using DFS, random, and a control-flow directed heuristic search algorithms. CREST relies on source level instrumentation, modifying the code to log information related to branches in the test program. The better performing heuristic search depends on the control flow information which is difficult to retrieve from binary programs.

JCUTE [31] is a Java symbolic execution engine that integrates different search algorithms.

DSE, the foundation of CREST, JCUTE and our DigEXE framework, has been widely used in automatic test case generation. The technique has evolved over the years, and more recently it has been used in the testing of dynamic web applications [32] and mobile applications [33]. SAGE [12] is a typical dynamic symbolic execution engine that is capable of detecting bugs in real world software. It performs

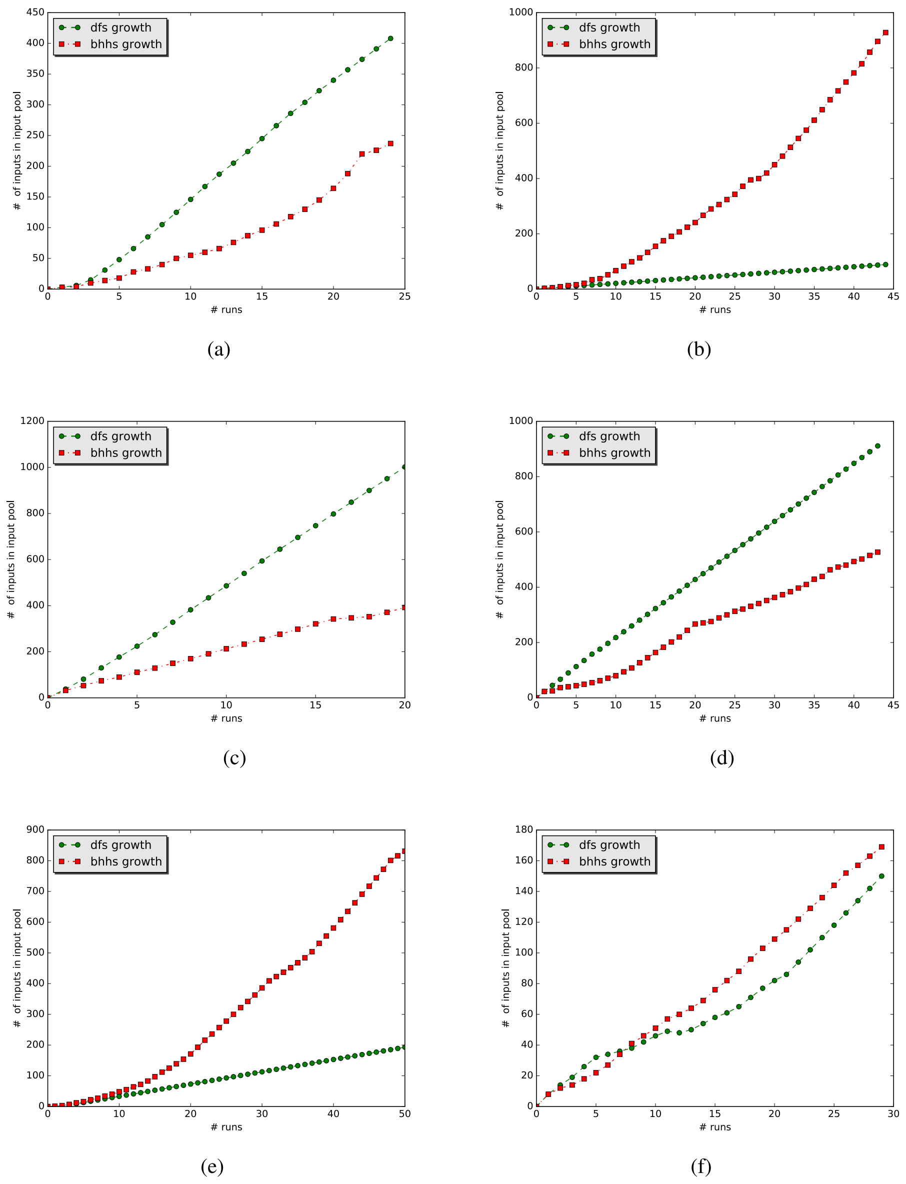


FIGURE 4. Input variety statistics of benchmark programs. (a) cjpeg. (b) convert. (c) js. (d) ld. (e) readelf. (f) tiffinfo.

optimizations mainly by looking at the syntactic features of path conditions in a program. SAGE uses branch condition flipping count as an hint for optimization purposes.

There have been other studies on the usage of runtime information to enhance the input space search process. DyTa [34] demonstrates the usage of static verification results to boost dynamic symbolic execution. In [11], length-k path is used to guide the searching algorithm in dynamic symbolic execution, and evaluate the mutant killing capabilities of the new method. The work in [35] addresses the problem of finding a path of a program which satisfies a regular property with guided dynamic symbolic execution.

However, our BHHS search algorithm works differently, that is improving search process by leveraging the refined execution history. While we would like to implement our proposed search algorithm in SAGE, the latter is not publicly available. Therefore, we compare the new BHHS search with an improved version of DFS search of the open source Fuzzgrind [19], which has very similar functionality with SAGE.

VIII. CONCLUSION

Software applications will play an increasingly important role in our data-driven society (e.g. in in Internet-of-Things environment and cyber-physical systems such as those found in smart cities and smart nations). In the rush to design software offering more complex features, it is highly unlikely to find a perfectly designed or bug-free software; hence, the need for effective bug detection approaches, bug reproduction productivity tools, and test case generation approaches.

In order to effectively generate high quality test cases, a branch history directed heuristic search was presented in this paper. We integrated the directed search algorithm within a dynamic symbolic execution engine. We also implemented a history directed depth first search, which is an improved version than the version of algorithm in the baseline DSE tool Fuzzgrind. We then evaluated our new BHHS search algorithm on a set of real world applications. The findings demonstrated that with the history directed heuristics, more branches can be covered with reduced solving time. Therefore, our method outperforms the improved DFS search.

In this paper, the input history and branch history were used to guide the search for high quality test inputs. However, the refined execution history could also serve as a database of program input and runtime data. We would also be able to extract useful features from the history information, combining other related statistics. Therefore, these features could be used to classify inputs into different clusters, model the program behavior for similar inputs, etc. This will form the basis of our future work.

REFERENCES

[1] M. Roopaei, P. Rad, and K.-K. R. Choo, "Cloud of things in smart agriculture: Intelligent irrigation monitoring by thermal imaging," *IEEE Cloud Comput.*, vol. 4, no. 1, pp. 10–15, Jan. 2017.

[2] Y. Kong, M. Zhang, and D. Ye, "A belief propagation-based method for task allocation in open and dynamic cloud environments," *Knowl.-Based Syst.*, vol. 115, pp. 123–132, Sep. 2017. [Online]. Available: http://dx.doi.org/10.1016/j.knosys.2016.10.016

[3] J. Xuan, H. Jiang, Z. Ren, Y. Hu, and Z. Luo, "A random walk based algorithm for structural test case generation," in *Proc. 2nd Int. Conf. Softw. Eng. Data Mining (SEDM)*, Jun. 2010, pp. 583–588.

[4] L. Zhang, B. Yin, J. Lv, K. Cai, S. S. Yau, and J. Yu, "A history-based dynamic random software testing," in *Proc. IEEE 38th Annu. Comput. Softw. Appl. Conf. Workshops (COMPSAC)* Vasteras, Sweden, Jul. 2014, pp. 31–36.

[5] R. Ramler, K. Wolfmaier, and T. Kopetzky, "A replicated study on random test case generation and manual unit testing: How many bugs do professional developers find?" in *Proc. 37th Annu. IEEE Comput. Softw. Appl. Conf. (COMPSAC)* Kyoto, Japan, Jul. 2013, pp. 484–491.

[6] V. Chipounov, V. Kuznetsov, and G. Candea, "The S2E platform: Design, implementation, and applications," *ACM Trans. Comput. Syst.*, vol. 30, no. 1, p. 2, 2012.

[7] P. P. F. Chan, L. C. K. Hui, and S. Yiu, "Heap graph based software theft detection," *IEEE Trans. Inf. Forensics Security*, vol. 8, no. 1, pp. 101–110, Jan. 2013.

[8] S. Naval, V. Laxmi, M. Rajarajan, M. S. Gaur, and M. Conti, "Employing program semantics for malware detection," *IEEE Trans. Inf. Forensics Security*, vol. 10, no. 12, pp. 2591–2604, Dec. 2015.

[9] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, San Diego, CA, USA, Dec. 2008, pp. 209–224.

[10] J. P. Galeotti, G. Fraser, and A. Arcuri, "Extending a search-based test generator with adaptive dynamic symbolic execution," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, San Jose, CA, USA, Jul. 2014, pp. 421–424.

[11] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," in *Proc. ACM SIGPLAN Int. Conf. Object Oriented Program. Syst. Languages Appl. (OOPSLA)*, Indianapolis, IN, USA, Oct. 2013, pp. 19–32.

[12] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, San Diego, CA, USA, Feb. 2008, pp. 1–16.

[13] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, pp. 1–10, 2008.

[14] N. Chen and S. Kim, "STAR: Stack trace based automatic crash reproduction via symbolic execution," *IEEE Trans. Softw. Eng.*, vol. 41, no. 2, pp. 198–220, Feb. 2015.

[15] J. Song, H. Kim, and S. Park, "Enhancing conformance testing using symbolic execution for network protocols," *IEEE Trans. Rel.*, vol. 64, no. 3, pp. 1024–1037, Mar. 2015.

[16] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. 2nd IEEE Int. Symp. Code Generat. Optim. (CGO)*, San Jose, CA, USA, Mar. 2004, pp. 75–88.

[17] *Crest*, accessed on May 26, 2017. [Online]. Available: https://code.google.com/p/crest

[18] X. Zhai *et al.*, "A method for detecting abnormal program behavior on embedded devices," *IEEE Trans. Inf. Forensics Security*, vol. 10, no. 8, pp. 1692–1704, Sep. 2015.

[19] *Fuzzgrind*, accessed on May 26, 2017. [Online]. Available: http://esec-lab.sogeti.com/pages/Fuzzgrind

[20] *Valgrind*, accessed on May 26, 2017. [Online]. Available: http://valgrind.org

[21] *Dynamorio*, accessed on May 26, 2017. [Online]. Available: http://www.dynamorio.org

[22] C. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. Conf. Program. Language Design Implement. (ACM SIGPLAN)*, Chicago, IL, USA, Jun. 2005, pp. 190–200.

[23] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," İin *P*roc. 19th Int. Conf. Comput. Aided Verification (CAV), Berlin, Germany, Jul. 2007, pp. 519–531.

[24] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun, "Locating need-to-externalize constant strings for software internationalization with generalized string-taint analysis," *IEEE Trans. Softw. Eng.*, vol. 39, no. 4, pp. 516–536, Apr. 2013.

[25] K. Chen, D. Feng, P. Su, and Y. Zhang, "Black-box testing based on colorful taint analysis," *Sci. China Inf. Sci.*, vol. 55, no. 1, pp. 171–183, 2012.

[26] R. Majumdar and R.-G. Xu, "Directed test generation using symbolic grammars," in *Proc. 22nd IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, Atlanta, GA, USA, Nov. 2007, pp. 134–143.

[27] B. Gu, V. S. Sheng, K. Y. Tay, W. Romano, and S. Li, "Incremental support vector learning for ordinal regression," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 26, no. 7, pp. 1403–1416, Jul. 2015.

[28] B. Gu, V. S. Sheng, Z. Wang, D. Ho, S. Osman, and S. Li, "Incremental learning for $\nu\nu$-support vector regression," *Neural Netw.*, vol. 67, pp. 140–150, Jul. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.neunet.2015.03.013

[29] Y. Zheng, J. Byeungwoo, D. Xu, Q. M. J. Wu, and H. Zhang, "Image segmentation by generalized hierarchical fuzzy C-means algorithm," *J. Intell. Fuzzy Syst.*, vol. 28, no. 2, pp. 961–973, 2015.

[30] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *Proc. Companion 22nd Annu. Conf. Object-Oriented Program., Syst., Languages Appl. (ACM SIGPLAN OOPSLA)*, Montreal, Quebec, Canada, Oct. 2007, pp. 815–816.

[31] K. Sen and G. Agha, "CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools," in *Computer Aided Verification* (Lecture Notes in Computer Science), vol. 4144, T. Ball and R. Jones, Eds. Berlin, Germany: Springer, 2006, pp. 419–423.

[32] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for javascript," in *Proc. 9th Joint Meeting Found. Softw. Eng. (ESEC/FSE)*, New York, NY, USA, Sep. 2013, pp. 488–498.

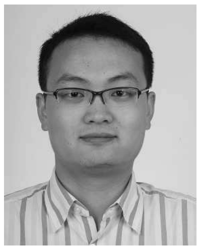
[33] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, New York, NY, USA, 2012, pp. 59:1–59:11.

[34] X. Ge, K. Taneja, T. Xie, and N. Tillmann, "Dyta: Dynamic symbolic execution guided with static verification results," in *Proc. 33rd Int. Conf. Softw. Eng. (ICSE)* Honolulu, HI, USA, May 2011, pp. 992–994.

[35] Y. Zhang, Z. Chen, J. Wang, W. Dong, and Z. Liu, "Regular property guided dynamic symbolic execution," in *Proc. 37th IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*, vol. 1. Florence, Italy, Sep. 2015, pp. 643–653.

YAN HU received the B.Sc. and Ph.D. degrees in computer science from the University of Science and Technology of China, China, in 2002 and 2007, respectively. He is currently an Assistant Professor with the School of Software, Dalian University of Technology, China. His research interests include model checking, program analysis, and security testing.

WEIQIANG KONG received the bachelor's and master's degrees in computer science from Wuhan University, China, in 2000 and 2003, respectively, and the Ph.D. degree in information science from the Japan Advanced Institute of Science and Technology in 2006. He is currently a Professor with the Dalian University of Technology, China. His research interests focus on formal methods, in particular, formal verification with hybrid model checking techniques for software analysis.



YIZHI REN received the Ph.D. degree in computer software and theory from the Dalian University of Technology, China, in 2011. From 2008 to 2010, he was a Research Fellow with Kyushu University, Japan. He is currently an Associate Professor with the School of Cyberspace, Hangzhou Dianzi University, China. His current research interests include: network security, complex data/network, evolutionary game theory, and trust management.

KIM-KWANG RAYMOND CHOO (SM'15) received the Ph.D. degree in information security from the Queensland University of Technology in 2006. He currently holds the Cloud Technology Endowed Professorship with The University of Texas at San Antonio. He was named one of 10 Emerging Leaders in the Innovation category of The Weekend Australian Magazine/Microsofts Next 100 series in 2009, and Cybersecurity Educator of the Year–APAC (Cybersecurity Excellence Awards are produced in cooperation with the Information Security Community on LinkedIn) in 2016. In 2015, he and his team won the Digital Forensics Research Challenge organized by Germany's University of Erlangen–Nuremberg. He was a recipient of the ESORICS 2015 Best Research Paper Award, the Highly Commended Award from Australia New Zealand Policing Advisory Agency in 2014, the British Computer Society's Wilkes Award, the Fulbright Scholarship in 2009, and the 2008 Australia Day Achievement Medallion. He is also a fellow of the Australian Computer Society.