Demo: A Symbolic N-Variant System

Jun Xu[†], Pinyao Guo[†], Bo Chen[‡], Robert F. Erbacher[*], Ping Chen[†], and Peng Liu[†]
[†]The Pennsylvania State University
[‡]University of Memphis
[*]Army Research Laboratory
{jxx13,pug132}@ist.psu.edu bchen2@memphis.edu
robert.f.erbacher.civ@mail.mil {pzc10,pliu}@ist.psu.edu

ABSTRACT

This demo paper describes an approach to detect memory corruption attacks using artificial diversity. Our approach conducts offline symbolic execution of multiple variants of a system to identify paths which diverge in different variants. In addition, we build an efficient input matcher to check whether an online input matches the constraints of a diverging path, to detect potential malicious input. By evaluating the performance of a demo system built on Ghttpd, we find that per-input matching consumes only 70% to 96% of the real processing time in the master, which indicates a performance superiority for real world deployment.

Keywords

Diversification; N-Variant; Symbolic execution

1. INTRODUCTION

To address threats from memory corruption vulnerabilities, an N-Variant system framework is proposed by Cox et al. [4]. As shown in Figure 1, an N-variant system works as follows. The polygrapher copies each input to N artificially diversified variants of the service process. The monitor compares the behavior of the variants to detect divergences which indicate attacks. The effectiveness of an N-Variant system is based on one insightful assumption, instead of security secrets. That is, no single malicious input can simultaneously compromise all the variants, though every variant can be vulnerable. They enforce this assumption via isolating variants into exclusive spaces.

However, there are two dominant barriers for broad adoption of N-Variant system. First, the amount of resources (CPUs & memories) consumed by an N-Variant system are N times as many as mere diversity [9, 12]. Second, run-time monitoring and synchronization of different variants are still required for N-Variant system to detect attacks, which introduces inevitable performance overhead. Evaluation of the prototype shows that the overhead ranges from 17% to 48%, depending on the diversification and the workload status of the server [4].

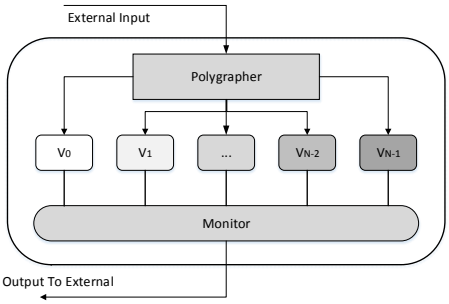

Figure 1: N-Variant system framework.

In this paper, we demonstrate a novel approach to build N-Variant systems. Instead of running multiple diversified processes to monitor divergences, we build a *divergence table* through offline analysis and convicts online inputs that match a record in the table. The divergence table holds conditions for *symbolic* paths that malicious inputs follow. At a high level, we do symbolic execution against the set of variants in a traditional N-Variant system, to get the execution paths from each of the variants. Paths diverging in different variants are selected to build the divergence table.

2. OVERVIEW

In this section, we define our threat model and provide a high level description of our technical approach. Before our overview, we define here some important terms used throughout the paper. We consider *"process = software + run-time environment"*, in which run-time environment mainly refers to memory management (including code and data layout, stack management, and heap management). If neither the software nor the environment is diversified, the process is called *master*; otherwise the process (running the same task as the master) is called a *variant* of the master.

2.1 Threat Model

In our model, a defender runs a master process P, and an attacker knows a zero-day memory corruption vulnerability V in P. The attacker launches an attack with payload A to exploit V. A compromises/crashes P. The security analysis team notifies the defender that A occurred and any mutations of A should be blocked. However, the team has not

MTD'16 October 24-24 2016, Vienna, Austria

ACM ISBN 978-1-4503-4570-5/16/10.

DOI: http://dx.doi.org/10.1145/2995272.2995284

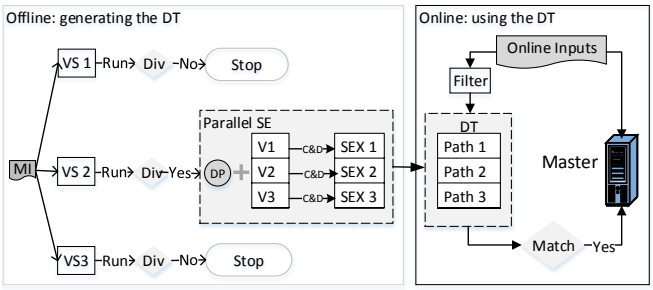

Figure 2: Workflow of our approach (MI: malicious input; VS: variant set; Div: Diverge; V: single Variant; SEX: Symbolic Executor; C&D: chopping and diversifying; DT: Divergence Table; DP: Diverging Point; SE: Symbolic Execution).

finished analyzing the vulnerability and hence no patch is available.

2.2 Technical Approach

We present the workflow of our approach in Figure 2. Our approach includes an offline and an online part.

The **offline part** aims to generate the divergence table with the presence of $\{P, V, A\}$. Generating the divergence table goes through several major steps listed in the following: (1) We first prepare multiple sets of variants. Variants in the same set are generated by diversifying the same aspect of the master, requiring that different variants have exclusive diversification spaces. Given the suspicious input A, we put it into each prepared variant set. If the input causes divergence among variants in a set, we take this set as a candidate. If multiple candidates are available, we randomly select one. (2) Given the selected variant set, we run A in each variant to locate the *diverging point*. Here we define diverging point as the last identical instruction in the executed path followed by A in different variants. (3) After the diverging point is located, we proceed to parallel symbolic execution. We customize a symbolic execution engine to support parallel yet synchronized execution. The engine will fork a separate executor responsible for each variant. Before starting symbolic execution, the executors chop the software to remove paths that cannot reach the diverging point. During symbolic execution, the executors will interact with others to synchronize the execution state. If a path diverges in different executors, it will be terminated and the engine will export the constraints for an input to follow this path. Each exported path as well as the corresponding constraints will be included as a record in the divergence table, which will be used for online detection of malicious inputs.

The **online part** uses the divergence table, which consists of two components: a master (providing services) and a searching engine. When an input arrives, the search engine matches it against the divergence table for attack detection (a match indicates that the input is a potential attack). Prior to the search engine, a heuristic-based filter can be designed to exclude benign inputs, which can help reduce the workload for the search engine [5].

3. IMPLEMENTATION

Our approach requires building the divergence table and using the divergence table in online detection. To build the divergence table, we customized KLEE to support the required features. More than 2,000 lines of `C++` code (including header files) are added to achieve the customization. To support online detection, we build a search engine to match input in the divergence table. The search engine is generated using a search engine generator written in more than 200 lines of `Java` code.

3.1 Customizing KLEE

We first provide an overview of KLEE and then present how we customize KLEE. KLEE is an open source symbolic execution engine for LLVM intermediate representation (IR) [7] of `C/C++` programs.

When all inputs are concrete, KLEE works as an interpreter. KLEE manages memory as a normal operating system does: (1) code and global variables are separated in different segments; (2) local variables are maintained on the stack and each function corresponds to a frame; (3) dynamically allocated objects are maintained in a heap implemented as a memory map; (4) access memory objects based on their addresses. To support calls to standard `C` library functions, KLEE links a copy of the LLVM IR of `uClibc` [2] with the interpreted program.

When symbolic inputs are fed in, KLEE will work as a symbolic execution engine. KLEE is a single-process and single-thread engine, so only one path can be executed at any time. Paths forked during execution will be pushed to a queue. For each path in the queue, KLEE maintains a private address space so that memory interference is not possible.

Feature 1: Memory model. The first feature KLEE needs is a memory model that can capture the semantics of memory corruption. KLEE uses a memory model that can largely approximate the model supporting native code. The only issue is that KLEE will conservatively check memory access. Memory corruption, such as buffer overflow, will directly stop the engine and raise an alert. To handle this issue, we loosen the memory check in KLEE. Currently, we only revised KLEE to support execution when buffer overflow occurs.

Feature 2: Incorporating the diversification. The second feature requires to incorporate into KLEE with diversification that creates the selected variant sets. Currently, we instrumented KLEE to support *address space partitioning* [4] and *reverse stack execution* [11].

First, we present how we revise KLEE to support address space partitioning. When interpreting a program, KLEE actually maps memory used by the interpreted program into the address space of KLEE itself, which means these memory cannot be reused until the interpreted program releases it. In this sense, if we start two executors to execute two program copies, the address spaces managed by the two executors are already largely isolated. We just need to prevent memory from being cross-used by the two executors.

Second, we explain the details of reverse stack execution with KLEE. In KLEE, the stack is implemented as a vector and each element in the vector corresponds to a frame. Each frame maintains an array of local variables for a function on the call stack. We manage a downward stack in this way: the first frame will be stored on the back of the stack vector and the first local variable in the current function will be stored at the last position of the current frame array; To the contrast, we manage an upward stack in the reverse way:

Table 1: List of check point.

check Point	State to Synchronize	Status to Compare
Conditional statements	Select the same branch to follow	Instructions executed along this path
Path forking	Manage the forked paths in the same way	The number of paths forked; The constraints added to each forked path
Symbolized address accessing	Concretize the address to the same value	Legitimacy of the address
Concretizing data	Concretize the data to the same value	NA
Path terminating	Select the same path to continue	Termination status; Path constraints
Unmarked path	Select the same path to continue	Unmarking status

the first frame will be stored on the front of the stack vector and the first local variable in the current function will be stored at the first position of the current frame array.

Feature 3: Parallel and synchronized symbolic execution. This feature is critical for our approach to work. Originally, to test software, KLEE creates an instance of the `Executor` class to work as an executor. The executor will load the LLVM IR of the software, prepare the run-time environment, and then start symbolic execution.

We instrumented KLEE to first create N threads. In each thread, we initialize an executor to couple with a member in the selected variant set, namely loading the IR version of the software in the variant and diversifying the run-time environment as the variant does. In particular, if the diversification is address space partitioning, we do not take any extra actions.

When an executor encounters a check point, as shown in Table 1, the executor will synchronize its state with others and check if its state diverges from others.

Feature 4: Diverging-point-specific path exploring. To support this feature, we implemented a LLVM pass [8] to confine the CFGs and built this pass into the customized KLEE. This pass first reuses built-in passes in LLVM to extract CG, intra-procedure CFGs, natural loops in the CFGs, and intra-procedure Data Flow Graphs (DFG) from the to-be-tested software. Specifically, our passe reuses (1) the `CallGraph` pass to get the CG; (2) APIs in `CFG.cpp` to get the CFGs; (3) the `LoopInfo` pass to get the natural loops (subloops are iteratively extracted); (4) an open source project [6] to get the DFGs. Then the pass will go through the aforementioned steps to unmark unreachable functions, redundant loops, and unreachable basic blocks.

3.2 Building the Search Engine

We serialize the constraint tree into indexed checking code blocks for input matching. Each code block checks the satisfiability of the constraints on a tree node and its sibling nodes, using an `if-else` statement. Starting from the block representing the root node of the constraint tree, upon satisfying a branch's constraints, the matching process jumps to another code block, until a block representing a leaf node is reached. The matching process returns the path that corresponds to an input if found, and 0 otherwise.

4. DEMO

4.1 Setup of Demo

In our prototype, the master runs a web server, `Ghttpd-1.4` [10] on a standard Linux environment. The server contains a buffer overflow vulnerability [1]. The vulnerability summary has been shown in Code Segment 1. We reduce the

Code Fragment 1 Vulnerable code in our running example

```
1  char SERVERROOT[255] = "/usr/local/wb";
2  void Log(char* request)
3  {
4      char logfilename[255];
5      char temp[200];
6      FILE *logfile;
7      sprintf(logfilename, "\%s/wb.log", SERVERROOT);
8      sprintf(temp, "Connection request: \%s", request);
9      if((logfile = fopen(logfilename, "at"))==NULL)
10     {
11       printf("Warning: Log Missing\n");
12       return;
13     }
14     fputs(temp, logfile);
15     fclose(logfile);
16 }
```

size of the `temp` buffer to be 30 bytes so that our symbolic execution can explore paths that trigger the overflow within an acceptable amount of time. We observe that a request P, which is a legit request longer than 30 bytes, crashed the system but was not logged.

We compile the source code of the web server into LLVM IR and then execute the IR in different sets of variants with P. As explained before, we select the two KLEE variants with different stack growing directions for symbolic execution. After the control flow is confined, we concurrently run the two KLEE variants for 48 hours with 60 bytes of symbolic input. As a result, 135,916 paths are identified as vulnerable. On average, we need 1.896 seconds to explore a single path. For each path, we generate one test example which satisfies all of its path constraints for evaluation purpose. We loaded these paths into a divergence table and developed a search engine for the divergence table.

Here we clarify why divergence will appear when the inputs are symbolized. When the input overflows the `temp` buffer, the `logfilename` in the variant with downward stack will be filled with symbolic values while the `logfilename` buffer in the variant with upward stack is not polluted. When `ling 9` is reached, the executor will open a symbolic file for the variant with downward stack but open a concrete file for the variant with upward stack. By default, KLEE will fork a path when a symbolic file is to be opened (one path for success and one path for failure). Thus, path forking happens in one variant but not in another variant, the customized KLEE will detect this divergence.

4.2 Evaluation Results

We evaluated the prototype in terms of run-time performance and detection efficiency. All our evaluations are com-

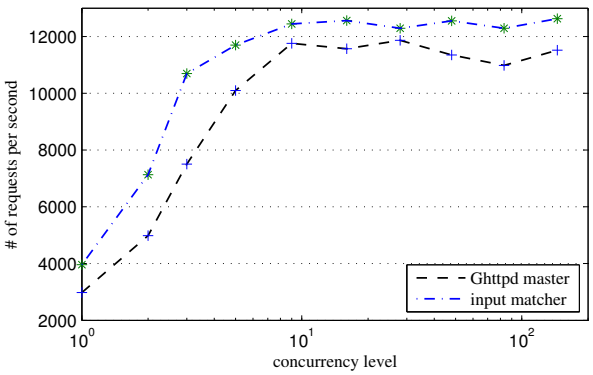

Figure 3: Performance comparison under different concurrency levels between the master and the input matcher in the DSE system.

pleted on an 8-Core 3.4GHz I7 machine with 8GB RAM running x86-64 RedHat.

Efficiency. We evaluate the efficiency of our DSE system in terms of time consumption for handling online inputs between the input matcher and the master with `Ghttpd-1.4` deployed inside the DSE system. We launch the evaluation using the ApacheBench v2.3 [3] web server benchmark. We customize ApacheBench to support sending different requests. Each request is assembled with data that corresponds to different paths extracted from the symbolic execution.

In the first experiment, we compare the performance of the master and our input matcher. We leverage ApachBench to send all 135,916 test inputs, which corresponds to 135,916 paths, to the master machine on different concurrency levels. Figure 3 shows the performance comparison. The results indicate that the input matcher performs better than the master, under both saturated and unsaturated workloads. The main reason is that our input matcher is file I/O free and thus completes most computation in memory. However, as a web server, Ghttpd is I/O intensive and must read and send target files. In the meantime, constraints extracted from symbolic execution only capture branch conditions in the program, which also eliminates other numerical operations inside the program.

Effectiveness. To evaluate the effectiveness of our DSE system, all 135,916 test cases are sent to the system for path matching. We found that all test cases are matched to their corresponding paths, which justifies the effectiveness of our system for potential attack input detection.

5. ACKNOWLEDGEMENTS

This work has been supported by ARO W911NF-13-1-0421. Bo Chen was supported by ARO W911NF-15-1-0576. Bo Chen would also like to thank the support from Center for Information Assurance at the University of Memphis.

6. REFERENCES

[1] CVE-2001-0820. Available from MITRE, CVE-ID CVE-2001-0820., Dec. 2001.

[2] E. Andersen. uClibc: AC library for embedded linux, 2005.

[3] Apache. ab - apache http server benchmarking tool, 2015.

[4] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: A secretless framework for security through diversity. In *Usenix Security*, volume 6, pages 105–120, 2006.

[5] P. Guo. Design, Implementation and Evaluation of a Symbolic N-Variant Simulator. Master's thesis, Pennsylvania State University, United States, 2015.

[6] Keutoi. llvm-dataflow-graphs. https://github.com/k3ut0i/llvm-dataflow-graphs, 2015.

[7] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[8] C. Lattner and V. Adve. Llvm language reference manual, 2006.

[9] H. Okhravi, M. Rabe, T. Mayberry, W. Leonard, T. Hobson, D. Bigelow, and W. Streilein. Survey of cyber moving targets. *Massachusetts Inst of Technology Lexington Lincoln Lab, No. MIT/LL-TR-1166*, 2013.

[10] G. Owen. ghttpd. http://gaztek.sourceforge.net/ghttpd/index.html, 2006.

[11] B. Salamat, A. Gal, and M. Franz. Reverse stack execution in a multi-variant execution environment. In *Workshop on Compiler and Architectural Techniques for Application Reliability and Security*, pages 1–7, 2008.

[12] J. Xu, P. Guo, M. Zhao, R. F. Erbacher, M. Zhu, and P. Liu. Comparing different moving target defense techniques. In *Proceedings of the First ACM Workshop on Moving Target Defense*, pages 97–107. ACM, 2014.