InputFinder: Reverse Engineering Closed Binaries using Hardware Performance Counters

Bogdan Copos
University of California, Davis
bcopos@ucdavis.edu

Praveen Murthy
Fujitsu Laboratories of America
praveen.murthy@us.fujitsu.com

ABSTRACT

The effectiveness of many dynamic program analysis techniques depends heavily on the completeness of the test suite applied during the analysis process. Test suites are often composed by developers and aim at testing all of the functionality of a software system. However, test suites may not be complete, if they exist at all. To date, only two methods exist for automatically generating test input for closed binaries: fuzzing and symbolic execution. Despite previous successes of these methods in identifying bugs, both techniques have limitations. In this paper, we propose a new method for autonomously generating valid input and identifying protocols for closed x86 binaries. The method presented can be used as a standalone tool or can be combined with other techniques for improved results. To assess its effectiveness, we test InputFinder, the implementation of our method, against binaries from the DARPA Cyber Grand Challenge example set. Our evaluations show that our method is not only effective in finding input and determining whether a protocol is expected but can also find unexpected control flow paths.

1. INTRODUCTION

Vulnerability testing is an expensive and labor intensive process. For many years, researchers have explored both static and dynamic methods for analyzing programs [9]. While static analysis can be more thorough, dynamic testing is often preferred due to its time efficiency. However, dynamic testing heavily depends on the availability of appropriate inputs or test suites.

Without complete test suites, the program cannot be properly tested and some bugs may be omitted during testing. While valid input can be learned from software documentation or source code, when such information is not available or complete, other methods must be used. Automated dynamic analysis techniques such as fuzzing and symbolic execution attempt to generate input for a given program. Currently these are the only two options available and both approaches present limitations. Fuzzing done randomly with no inference attempted of the program's behavior cannot be effective as the input space is usually too large. Symbolic execution has traditionally been used with manual intervention to identify potential symbolic variables to begin exploration with, a luxury we do not have when analyzing closed binaries autonomously, in addition to scalability issues and limitations of decision procedures that are used to solve the constraints.

In this paper, we introduce a new method and its implementation, called InputFinder, for generating valid user-input for closed binaries. This new method could be useful in autonomous vulnerability scanning systems, for reverse engineering unknown binaries, or as a complement to fuzzers or symbolic execution engines. Our method is composed of two main components. The first component exploits hardware performance counters to build valid input for closed binaries. Specifically, as the program is given various inputs, InputFinder records the number of instructions **retired** during the program's execution for each input and uses a statistical measure to infer the program's expected input. The number of instructions retired differs from the number of instructions executed by a processing unit. Because of the out-of-order CPU pipeline, a CPU may execute more instructions than necessary (e.g. as a result of wrong branch prediction). The instructions retired represent the subset of instructions executed which leave the *Retirement Unit* once execution has been deemed correct (i.e. the instructions which actually impact the program). Since our input finding method works by leveraging information about how much work the processor performs during input validation, it does not depend on a particular platform, format (i.e. binary, source, etc), or method of obtaining input (e.g. stdin, arguments, network socket, file IO). However, there are some constraints. The underlying method does not work if the user input is transformed (e.g. hashed) before validation. The implementation of our method also has some additional drawbacks. Our current implementation can only handle programs whose input is interpreted as a string (or a set of characters). The user input can have many components, however, there may not exist any internal dependencies between the components (e.g. value of one component defines the length of another component). If the input is composed of multiple components, the program must process the components sequentially in order for our method to work. The same hardware counters are used by InputFinder to also find the expected input size and to categorize the expected input based on type. The second component uses the discovered input to build a

PPREW-5, December 08 2015, Los Angeles, CA, USA

DOI: http://dx.doi.org/10.1145/2843859.2843865

protocol state machine associated with the tested program. This enables the generation of a more thorough test suite.

The main contributions of this paper are as follows:

- We introduce a new approach to automatically generate input valid strings for unknown, closed-source binaries by leveraging hardware instruction counters. Our method relies on observing changes in the number of instructions retired by programs during their input validation process to learn valid input strings.
- We describe how our method can be used not only to find valid input strings, but also to determine the expect input size and classify input based on type (e.g. alphabetic, alphanumeric, numeric, etc.).
- We develop a method for generating protocol state machines using the valid input previously found.
- We implement the techniques presented as a proof of concept tool and evaluate it using binaries published by DARPA as part of the 2014 Cyber Grand Challenge competition [5]. Additionally, we compare our approach to the manual process used by a security specialist to crack a binary as part of a job interview [6].

The paper is organized as follows. Section 2 discusses related works and highlights differences between the related works and the work presented here. Section 3 covers background information relating to concepts utilized in this work. The details of our approach are explained in Section 4. First, the method for finding valid input is described. Then we introduce our method for building protocol state machines using the discovered input. Section 5 describes the evaluation of the work presented and the results. In Section 6, we share future plans for our work and explain cases which the proposed method does not handle.

2. PREVIOUS WORK

Dynamic analysis is extensively studied in academia and is used widely in the industry by security specialists. Two of the main approaches to vulnerability testing are fuzzing and symbolic execution.

Fuzzers are a popular choice amongst software testers. They are easy to use, easy to implement, and can produce surprising results [12, 13]. There are two classes of fuzzers: black box fuzzers and white box fuzzers. In black box fuzzing, the program is watched as it is tested with randomly generated input. If the program crashes, the input(s) are recorded and reported. No prior program specifications or knowledge of the input format is available in such cases. Since black box fuzzing is completely random, its effectiveness is limited, especially with respect to coverage.

Symbolic execution is another method used for generating valid input. When a program analyzer symbolically executes a program, it does not use concrete values. Instead, the input is represented as a mathematical formula composed of symbols and equations resulting from the constraints (related to the input) gathered during the program's execution. Specifically, the tester first marks the input as symbolic. This can also be done automatically via taint analysis. The program is then executed by an interpreter that assumes symbolic values for those symbolic input locations. As the program executes, whenever operations are performed on those symbolic variables, a formula in terms of those symbols keeps track of the mathematical operations performed on the symbolic variables. The formula can also have constraints on those symbols based on outcomes of conditional branches encountered during the executions. At the end of execution, or during certain portions of the execution, the formula is examined by a constraint solver to determine whether it is satisfiable (meaning that a solution exists to make the formula true), and if it does, concrete values for all symbolic variables in the formula are computed. Those concrete values will then constitute a set of test cases that can execute the program along the set of paths (because of branches) encountered in the formula. Symbolic execution has been proven effective by previous works [8, 3]. One of the most important works in this area is described in [3] which introduces EXE, a dynamic analysis tool capable of generating inputs that crash programs by running the programs with symbolic input.

White box fuzzing leverages the symbolic execution traces of a given binary to produce more inputs. One of the many previous works is [7] which introduces SAGE, an automated white box fuzzer. White box fuzzing differs from black box fuzzing in that it uses the concrete inputs to gather constraints and then systematically negates constraint clauses to generate additional inputs that cover more branches in the program. White box fuzzing can attain better code coverage but can be slow. Symbolic execution (and consequently white box fuzzing) is crippled by innate limitations of constraint solvers. While symbolic execution engines perform well with linear constraints, non-linear constraints greatly impact efficiency. One of the works utilizing white box fuzzing is [14] which introduces SmartFuzz, a dynamic test generation tool. SmartFuzz uses symbolic execution to learn about the program's input and passes that information to a fuzzer for test case generation.

The main difference between our work and SmartFuzz is that our work does not use symbolic execution to gain insight into the format of the input. Similar to SmartFuzz, the output of our work can be used by a fuzzer to generate test cases. Unlike the EXE symbolic execution engine, the SAGE tool, and SmartFuzz, the approach presented in our work is not hindered by constraint solvers. Additionally, compared to EXE, our method does not require any instrumentation or source code. InputFinder does not execute the program with symbolic input but rather it bootstraps itself by building up a set of valid inputs (from an initially empty set) by gathering information about the execution of a program.

Another related work is presented in [9], where the authors use a combination of static and dynamic analysis to build a smart fuzzer. Their work uses static analysis to collect preliminary information such as an approximation of the inter procedural control flow graph. Having gathered static information, their dynamic analysis engine executes the program while monitoring dependencies between the input and the dynamic control flow path experienced by the program. Their work also takes advantage of a constraint solver that, given a path of the program, generates input which forces the program to follow the chosen path. Our work InputFinder differs from their approach in several ways. One difference is that our work uses neither static analysis nor constraint solvers. Instead our input finding method relies only on information extracted from the execution of the un-instrumented program.

Another area of research relevant to this paper is protocol reverse engineering. An important result in this area is [4] which introduces a technique for extracting protocol specifications from packet captures and automatically generating input which can be used by a stateful fuzzer. There are two notable differences between Prospex and our work. While the end result is similar, InputFinder assumes no prior knowledge about the tested program and hence it does not utilize packet captures for protocol specification extraction. Instead, InputFinder infers the protocol through dynamic analysis by automatically finding input and using that to generate a protocol state machine. Caballero et. al introduce Polyglot [2], a tool that uses dynamic binary analysis to observe how programs process the received application data and uses that information to infer the protocol's message format. Their work is similar because of the use of dynamic binary analysis and the use of information from program traces to make inferences about the protocol format. However, our work does not try to determine the protocol message format but rather the high level protocol. Additionally, unlike Polyglot, our work does not use test cases or previously known input. To date, we do not know of any protocol reverse engineering techniques which work without any prior knowledge about the program's input.

3. BACKGROUND

3.1 Hardware counter registers

Most modern microprocessors are equipped with a set of special-purpose registers used to count hardware events. Each register, or counter, can be programmed to measure specific events such as cache misses, floating point operations, and even instructions retired. Such hardware performance counters are often used by software developers to analyze their programs and improve the running time and efficiency. However, in this paper we show how such performance counters can also be used for reverse engineering. Utilities, such as Linux/Unix *perf*, take advantage of such hardware counters in order to provide developers with useful information about a program's performance. The *perf* utility allows developers to specify the desired hardware event they wish to measure. Once events have been selected, *perf* interacts with the appropriate kernel modules to program the hardware counters accordingly. In this work, we use *perf* to count the number of **user-land** instructions retired as a program is tested with various input. **User-land** instructions are instructions of a program which run outside the operating system's kernel. While other options are available for counting instructions, including program instrumentation, we choose *perf* for its efficiency and minimal overhead. Experimental results comparing *perf* and *Pin* instrumentation show that instrumentation is at least 2 times slower. The number of *total* instructions retired varies significantly for a binary from execution to execution [16]. This phenomenon reflects fluctuations in instructions retired by the kernel depending on the state of the machine due to cache-misses and other factors. Since we rely on this count, such variations are problematic. However, experimentally, the number of *user-land* instructions only varies, on average, by one instruction, giving us a reliable counter.

3.2 Execution/Dynamic Matching

In many application areas, including clone detection and software debugging, there is a need for comparing executions of a program (or multiple versions of a program). One of the proposed approaches aimed at tackling this issue is dynamic matching. Dynamic matching works by first collecting execution traces of different executions. Such traces can be collected through a number of ways. One option includes the use of a tool such as *Pin* [10]. *Pin* is a framework for building customized binary analysis tools and can be used to design a tool which creates a transcript of the instructions retired by a program. Once the traces are collected, they are analyzed and mappings between instructions of different executions are produced. These mappings can be used to understand the relationship between two executions (or two versions of a program). In this work, we rely on the concept of dynamic matching and the *Pin* framework to develop a protocol state machine generator for closed binaries.

4. METHODOLOGY

InputFinder is composed of two main modules. The first module is responsible for discovering input and properties of the expected input, including input size and the type of the input. The input generated by the first module is then used by the second module to generate a protocol state machine. The protocol state machine can be used by third party tools such as stateful fuzzers for even better code coverage.

4.1 Finding Valid Input

The majority of software programs accept input, perform transformations on the input, and output results. However, most programs do not accept completely random input. Input usually passes through an input validation filter. Such filters are snippets of code responsible for distinguishing between valid or good input (i.e. input the program was constructed to understand and accept) and bad input (i.e. input which is not useful or does not follow the desired format). Although validation mechanisms can be very sophisticated, they are often a combination of string comparisons and conditional statements. Code snippets of input validation from our tested binaries can be found in Appendix A. Our method exploits changes in the number of user-land instructions retired during multiple executions with varying inputs to make inferences about the program's validation mechanism. These changes reflect different execution paths of the program as a response to both valid and invalid input. Assuming most input is invalid, our method observes differences in the number of instructions retired to detect input which passes the validation mechanism.

Figure 1 shows the overall architecture of the finding input component of InputFinder.

The steps of the process for finding valid input are described in algorithm 1 and outlined below. Starting with an empty string, s:

1. InputFinder executes the program once for every printable character and records the number of user-land instructions retired by the program with the given input. The input is a string concatenation of the string s and the current printable character. The Unix *perf* utility is used for monitoring the number of instructions retired. Note that InputFinder does not record the total number of instructions but rather user-land instructions. The reason for this, as explained in Section 3.1, is in contrast to user-land instructions, that the

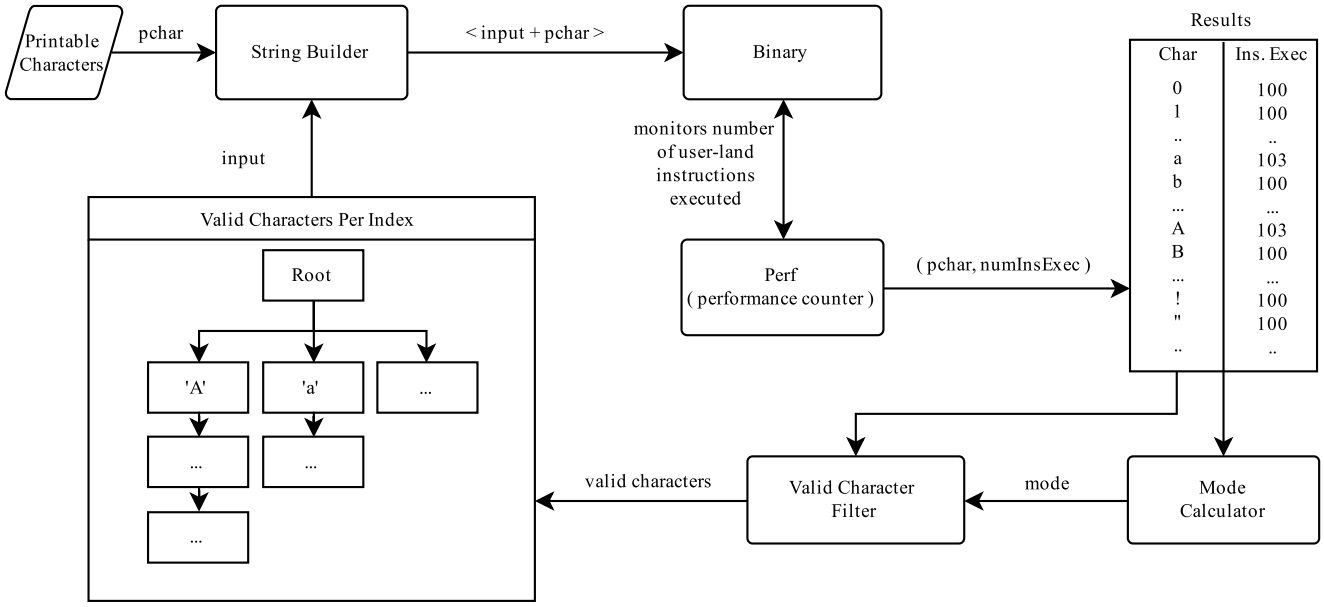


Figure 1: This figure depicts the architecture of the finding input process. There are three main parts: the input string builder, the performance counter (i.e. *perf* utility), and the valid character filter

Algorithm 1 Pseudocode implementing the input finding process

```
 1: s ← “”                                  ▷ starting with an empty string
 2: procedure FindInput(s)
 3:     for i=0, size(printableChars) do
 4:         char = printableChars[i]
 5:         insRetired = ExecBin(s + char)
 6:         insPerInput[char] = insRetired
 7:     end for
 8:     mode = CalcMode(insPerInput)
 9:     for i=0, size(printableChars) do
10:         char = printableChars[i]
11:         count = insPerInput[char]
12:         if mode-epsilon < count < mode+epsilon then
13:             ignore
14:         else
15:             validCharList += char
16:         end if
17:     end for
18:     if no valid characters found then
19:         end of string reached
20:         return
21:     else
22:         for i=0, size(validCharList) do
23:             validChar = validCharList[i]
24:             FindInput(s + validChar)
25:         end for
26:     end if
27: end procedure
```

total number of instructions retired can vary greatly from execution to execution for various reasons (e.g. branch prediction, cache). The Expect tool is used for automating the program interaction. This step is shown in line 5 of algorithm 1.

2. Once all executions have been completed (one for each character), the mode number of instructions retired is computed across all executions. The mode is defined as the value that appears most often in the set of recordings. This is done in line 8 of algorithm 1.

3. Next is the filtering stage which is responsible for identifying characters likely to be part of a valid input string at the current index. Under the assumption that most characters are not valid for the current index, InputFinder gathers all characters with the number of instructions retired outside the range of the mode +/- an epsilon value, as shown in lines 9-17 of algorithm 1. These characters represent valid characters for the current index of the input string. The epsilon value accounts for small variations in the hardware counters. While the initial reaction is to expect a valid character to result in more instructions retired than a non-valid character, this is not always the case. As discussed earlier, the input validation mechanism can be complex and can vary in behavior. The validation mechanism may verify input against all accepted input strings before denying the provided input. Such a scenario results in more instructions retired for invalid input than valid input.

4. For each valid character c, InputFinder forks, with every child process repeating steps 1-4, starting with $c+s$ as the initial input string, where + signifies concatenation (starting at line 22 of algorithm 1). If no valid

characters are found, the end of a valid input string has been reached and the string is recorded. If the current index is 0 and no valid characters have been found, the given binary has no predefined input commands.

Figure 2 depicts the results of the above process over the span of five consecutive time stages. As seen in the figure, at time *T0*, InputFinder begins with an empty string. At time *T1*, InputFinder finds valid characters for index 0 of the input string and is forked into n child processes where n is the number of valid characters found. This continues until all child processes are done.

4.1.1 Finding Desired Input Size

A similar approach can be used to detect the expected size of the program's input or even the size of the program's input buffer. Specifically, there may be a change in the behavior of the program with respect to the number of instructions retired when the size of the input is larger than expected. Most programs are designed to read a predefined number of bytes from the source of input and store the bytes in a buffer. However, the change in behavior as the input grows beyond the expected length can be observed and exploited. Usually, as the input size grows, the number of instructions retired also changes (increases usually, except if optimizations are in place). This happens because additional instructions are needed to read more characters from the input source. However, when the input string is larger than the expected size, the number of instructions retired no longer increases as a function of the input size. It should be noted that the CGC binaries do not use the standard library functions. Instead, the CGC platform implements their own functions for handling user input, which use the CGC platform system call *receive* to read one character at a time from the input source, in a loop, until a delimiter or a maximum size is reached. While the above described behavior is easily observed with CGC binaries, it also extends to programs using the standard GLIBC functions. For example, when using the GLIBC functions such as *getline* and *fgets*, and the user input exceeds the expected length, the number of user-land instructions retired changes and remains constant as long as the input length exceeds the expected length. This behavior can be observed and exploited to find the expected input size, which may often also be the size of the input buffer. The same behavior is exerted for other similar functions Our method observes such differences in the number of user-land instructions retired as the program is given increasingly larger input.

To start, InputFinder finds the set of all invalid characters for index 0 of all input strings and picks one at random. The character selected is used to build invalid input strings of various lengths. It is crucial to use invalid input to avoid noisy fluctuations in the number of instructions retired due to valid input processing. Initially, the program is executed twice: once with an input string of two invalid characters and once with an input string of three invalid characters. The number of user-land instructions retired during these two executions are recorded. These two values may be equal or different, depending on the program's logic.

The two values may be equal if, for example, the program uses the *read()* system call directly to read data from the input file descriptor. Since we are measuring the number of user-land instructions retired (not also instructions retired by the kernel), the instructions retired by the *read()* function are not included in our counter so there will be no linear increase in the number of instructions retired as the input size is increased. If the two values are equal, the program is executed with increasingly large input until a change in the number of executions is recorded.

If the values differ (as discussed earlier), the difference in the number of instructions retired is recorded and the program is executed, increasing the size of the input, until a change is detected. In most cases, the recorded difference represents the number of instructions needed to read an additional character. Imagine a program which reads one character at a time, in a loop, until the expected number of characters is read. The process stops when the difference in the number of instructions executed between two inputs of two consecutive lengths (i.e input X and input Y are of lengths n and $n+1$) is different from the recorded difference. It is also possible that the number of instructions retired oscillates between two (or more values). This can happen if the program processes inputs of various lengths differently. One example is a palindrome finder, where the execution paths for input of an odd length and input of an even length differ with respect to the number of instructions. The same underlying concept can be applied to such cases, where the two (or more) oscillating values are recorded, and the input string is increased until a difference is observed.

To speed up this process, instead of increasing the input size linearly, the input size is increased exponentially until a change is detected. Once the change is detected, the input size is decreased to find the exact size at which the change happens.

Determining the exact size is useful when performing symbolic execution on a buffer where one cannot make the size of the buffer symbolic (since frameworks such as *Fuzzball* do not allow for that). In such cases, being able to determine the buffer size is necessary and useful to instruct the symbolic execution engine the exact size of the buffer that should be made symbolic. We detected 5 crash inputs in the tested binaries because of our finding input size approach.

4.1.2 Categorizing Input

Our method is also capable of classifying the input discovered. Generated input strings are classified in three categories: alphabetic, numeric, alphanumeric. Similarly, the classification algorithm detects special characters. Once enough input strings have been identified, InputFinder uses the categorization data to generalize about the type of input the program accepts. This is done by iterating through the characters of the input strings discovered and determining which category they fall into. Such information is valuable especially in cases where the inputs discovered are all numeric or contain (one or more) special characters. Having learned such information about the program, the testing platform (e.g. fuzzer) can be adjusted to proceed accordingly.

4.2 Generating Protocol State Machine

Thus far we have described how InputFinder discovers valid input for a given binary autonomously. However, some binaries have an implicit protocol. Consider, for example, a database management program. Without first selecting a database and perhaps even a table, some commands such as *INSERT* or *SELECT* (or their equivalent) have no effect. We develop the second component of InputFinder to address this issue. Specifically, this component identifies expected

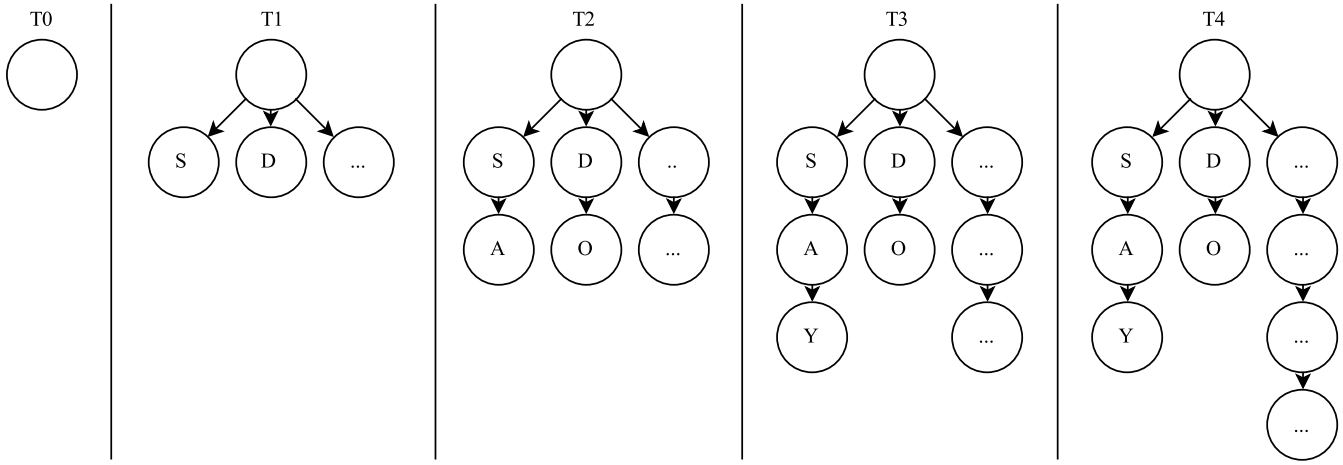
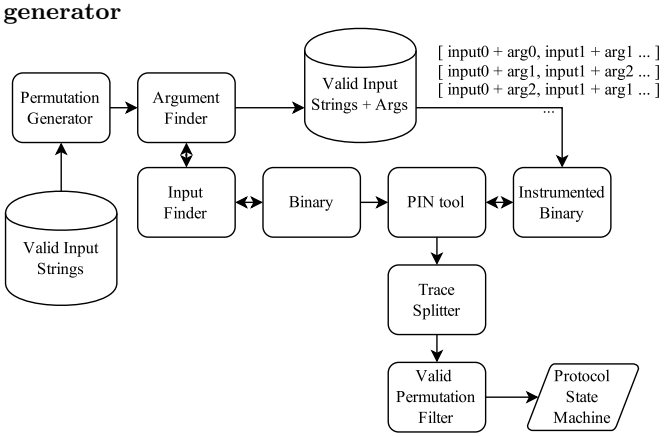


Figure 2: This diagram visualizes the input strings being constructed throughout five consecutive time stages of the execution of InputFinder. Multiple outgoing arrows represent a process forking. This occurs when multiple valid characters are detected for a given index of the input string and, as a result, the finding input process forks to concurrently handle all of the discovered characters.

orderings (or permutations) of the discovered input strings, and constructs a protocol state machine.

Figure 3: Architecture of the protocol state machine generator



The architecture of this component is depicted in Figure 3. The process is described below and shown as pseudocode in algorithm 2. The process can be split into three steps.

- First, all of the possible combinations of user inputs (given the discovered input strings) are generated.
- Then, the inputs are passed to an instrumented version of the binary, which outputs a single execution trace for each execution (one execution per permutation of input strings).
- Last, to determine the expect protocol, the execution traces are compared.

To generate all of the possible combinations of user inputs, all permutations of the discovered input strings are enumerated. The permutations range in size (i.e. number of strings) from 2 to n, where $n = |input\_strings\_set|$. However, enumerating all possible permutation of the discovered input strings may not be enough. Consider cases where the user input is composed of two (or more) strings where the first string is a command and the rest are arguments. Now also imagine that the arguments of an input string depend on the previously entered string. To cover such cases, for each permutation, we test each input string for arguments (using the previously mentioned input finding technique). For example, for a given permutation $\{AUTH, SET\}$, our tool will first try to identify arguments for the $AUTH$ input string. Without finding the correct argument for the $AUTH$ command, the execution of SET will not affect the program's state. Assume our approach was successful in identifying a single argument, *0*, for the $AUTH$ string. Now, the tool will attempt to find arguments for SET but with $AUTH$ *0* as the first entered user input. Again, this ensures that our approach identifies arguments for strings which depend on previously entered strings.

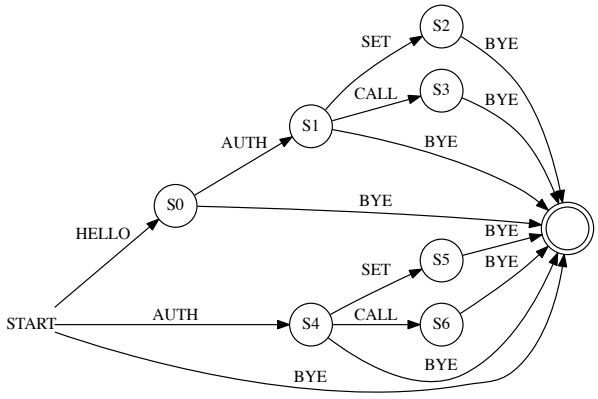
The resulting permutations of strings (and their arguments) are also passed through the instrumented binary in order to collect execution traces. The binary is instrumented with a *Pin* tool, developed by us, which logs the starting address of every basic block executed, as a function of time. A basic block is a set of instructions with a single entry and a single exit point. Once all permutations of a given size have been executed, the dynamic similarity test analyzes the resulting execution traces. Each execution trace is split into sections, each sections associated with the program processing one of the input strings of the input permutation. These sections are delimited by a short sequence of specific basic blocks which are executed right before the reading and processing of the user input. To determine the *correct* or *expected* input string permutation, the execution trace section of the last command in a permutation is compared against the execution trace sections of that same command, in the same index of the permutation. Under the assumption that

Algorithm 2 Pseudocode implementing the protocol state machine generator

```
1: procedure MAIN(inputs)
2:     for i = 1, SIZE(inputs) do
3:         length = i
4:         permutations = GETPERMUTATION(length)
5:         results = ∅
6:         for j = 0, SIZE(permutations) do
7:             permutation = permutations[j]
8:             EXEC(permutation, 0, results)
9:         end for
10:        COMPARE(results, inputs)
11:    end for
12: end procedure
13: procedure EXEC(cmds, i, results)
14:    if i == SIZE(cmds) then
15:        results += cmds
16:        return
17:    end if
18:    initCmds = cmds[0:i]
19:    cmd = cmds[i]
20:    args = FINDARGS(initCmds, cmd, i)
21:    for j=0, SIZE(args) do
22:        cmds[j] = cmds[j] + arg
23:        EXEC(cmds, i + 1, results)
24:    end for
25: end procedure
26: procedure COMPARE(results, inputs)
27:    for i=0, SIZE(results) do
28:        cmds = results[i]
29:        lastCmd = cmds[SIZE(cmds)-1]
30:        trace = EXECBINWITHPIN(cmds)
31:        traceForLastCmd = SPLITTRACE(trace, cmds)
32:        allTraces[lastCmd][cmds] = traceForLastCmd
33:    end for
34:    for i=0, SIZE(inputs) do
35:        cmd = inputs[i]
36:        for j=0, SIZE(allTraces[cmd]) do
37:            cmdTraces = allTraces[cmd][j]
38:            clusters = COMPARE(cmdTraces)
39:            goodPermutation = FILTER(clusters)
40:        end for
41:    end for
42: end procedure
```

the execution of the last command will exhibit a different behavior given the correct prior user inputs, our approach identifies which input string permutation causes the execution trace section of the last command to differ from other execution sections of the same command, in the same index of the permutation. If such a permutation is discovered, it is added to the protocol state machine.

Figure 4: Example of protocol state machine for one of the DARPA Cyber Grand Challenge example binaries.



An example of a protocol state machine can be seen in Figure 4. Since there is no information about the states and their meaning, the states are represented by circles with generic names (e.g. $S0$). The arrows represent input strings that enable the transition between states.

5. EVALUATION AND RESULTS

5.1 Finding Input, Input Size and Classifying Input

In order to determine its effectiveness, InputFinder was tested on 24 x86 binaries from the DARPA Cyber Grand Challenge Example set [5]. The Cyber Grand Challenge (CGC) binaries are 32 bit x86 binaries in the CGC format. The only difference between the CGC format and the well known ELF format is in the header of the binaries and in the fact that dynamic linking is not supported. Additionally, they are designed to only be executed in the Cyber Grand Challenge platform and utilize different system calls than the UNIX system calls. The Cyber Grand Challenge organizers also provide the source code of the binaries which, with some effort, can be modified to use UNIX system calls and be built as ELF x86 binaries. At the time of this writing, there were only 24 CGC binaries available, all of which were used for the analysis of our approach.

The results are summarized in Table 1. Analyzing the source code showed that out of the 24 binaries, 21 had predefined valid input strings. InputFinder found input strings for 13 of the 21 binaries and categorized the input correctly for 100% of the 13 binaries. The running time varied between a few minutes and an hour, depending on the number

and length of the valid input strings discovered. Our method was able to determine the input length for 19 of the 24 binaries. Note that our approach was able to find the expected input length even if it was not able to find valid input. Additionally, our method was able to determine crash inputs for 5 of the binaries. In all cases, the crash input caused a buffer overflow and caused the binary to exit unexpectedly.

Additionally, our input finding method was tested on a binary used for technical interviews. The binary was designed as a reverse engineering challenge for interviewees. To pass the challenge, the interviewee has to crack the secret passcode to the program. A blog post describes the manual process used by one interviewee to crack the password [6]. The manual process is tedious and it includes methods such as disassembling the binary, stepping through the instructions using a debugger, and reverse engineering the logic of the program. As described in the blog post, the cracking of the binary can take several hours and requires advanced skills and years of background knowledge. InputFinder was able to crack the binary in a matter of minutes autonomously.

5.2 Protocol State Machine Generation

The evaluation of the protocol state machine generator was also done using the DARPA Cyber Grand Challenge binaries. However, out of the total of 24 binaries, only 2 had a clear implicit protocol. Specifically, these 2 binaries requested authentication before allowing full functionality. Our method was able to generate a correct protocol state machine for both of these binaries. The protocol state machine generated was manually verified against the source code of the binaries.

5.3 Case Study

To further prove the usefulness of the method presented in this paper, we use one of the Cyber Grand Challenge binaries (i.e. 250d1101) as a case study and for comparison with a black-box fuzzer and a symbolic execution engine. The binary implements a simple protocol that lets the user call functions applying root64 and parcour schemes. The binary is built such that after sending the *HELLO* command, it will generate an authentication token which is echoed to the user and must be used for the *AUTH* command. Without proper authentication, the *SET* and *CALL* commands have no effect. Before generating the protocol state machine, InputFinder analyzed the binary and discovered the input strings $HELLO, AUTH, SET, CALL$, and BYE in approximately 20 minutes. These inputs are the only inputs implemented by the binary's protocol. The protocol state machine generator not only determined that authentication is necessary but also discovered that a user can properly authenticate themselves with a value of 0 or any special character if they omit the *HELLO* command (Figure 4). This shows that the input finding approach in combination with the protocol state machine generator can help identify potential backdoors or unexpected control flow paths for a given program.

When comparing our method with a black box fuzzer and a symbolic execution engine, we ran each technique for 30 minutes and gathered code coverage statistics. The black box fuzzer used is the open source *zzuf* fuzzer and the symbolic execution engine used is that of the *Fuzzball* [1, 11]. While most popular for its symbolic execution, *Fuzzball* is a tool for binary analysis, built on top of the BitBlaze Vine [15], which can do a number of things including extracting various traces and computing coverage statistics. For simplicity reasons, we use the *Fuzzball* tool to collect code coverage statistics. For compatibility reasons with the *Fuzzball* tool, we translated the source code of the binary to use UNIX system calls and compiled it in the ELF format. The changes are minor and have no impact on the functionality of the program.

The results of the comparison show that *zzuf* covered a total of 8936 basic blocks and was unable to discover any valid input strings. At the same time, the *Fuzzball* symbolic execution engine executed a total of 9922 and 10003 basic blocks (explained below) and was able to discover the valid input strings. The method presented in this paper executed 10055 basic blocks and not only discovered value input strings but also identified that a user can authenticate themselves with the command *AUTH 0*.

It should be noted that while the fuzzer and our method worked autonomously, the *Fuzzball* symbolic executioner required some reverse engineering of the program to help setup. One of the important parameters needed for symbolic execution was the address (and size) of the memory desired to be treated as symbolic. The different values of the basic blocks executed by *Fuzzball* are due to multiple executions of the engine with varying parameters. Overall, the results prove that the method presented in this paper is efficient with respect to both time and code coverage.

6. FUTURE WORK AND LIMITATIONS

While the method presented in this paper has many advantages, it also has a few limitations. The input finding implementation presented in this paper is only capable of finding input for programs which interpret the user input as a string (or set of characters). The method presented cannot handle cases in which input is composed of multiple interdependent fields. As an example, the method presented fails in cases where the input is composed of multiple fields and the value of a field determines the length of another field or in cases where binaries validate input fields out of order. Such failures were observed in the evaluation of InputFinder. Another reason of failure is binaries which accept non printable input (e.g. hex characters, binary, etc.).

Another drawback of the presented method is its runtime complexity. For any given binary, to find valid input strings (without valid arguments), the runtime is $O\{n*m\}$, where n is the length of the **longest** valid input string of the valid input string set and m is the total number of valid input strings accepted by the program. The process can be parallelized but it is limited by the number of hardware performance counter registers.

In a few cases, experiments have resulted in false positives, especially when experiments were run consecutively without pauses in between. Closer analysis showed that these false positives occur because of inaccurate performance counters, a phenomena we cannot yet explain. To account for this, an experiment can be re-run.

The method proposed in this paper is not the only way of finding input strings for a closed binary. Similar results may be obtained by instrumenting the binary (using *Pin* or even a debugger such as the *GNU GDB*) to count the number of instructions executed. However, there is a significant difference in the overhead and runtime between instrumentation and our approach. This is one of the reasons why for the

Binary	Input	Number of Inputs	Input Size	Crash Input	Protocol State Machine
06459301	yes (8 min)	8 out of 8	yes	no	-
06b71301	no	-	yes	no	-
07a9a901	-	-	yes	yes	-
0b32aa01	yes (2 min)	1 out of 1	yes	yes	-
11dc8e01	no	-	yes	no	-
1877a601	yes (11 min)	6 out of 6	yes	no	-
250d1101	yes (20 min)	5 out of 5	yes	no	yes
2eca0101	yes (42 min)	7 out of 7	no	no	-
37e97201	yes (38 min)	6 out of 6	yes	no	-
3dcf1a01	yes (6 min)	6 out of 6	yes	yes	-
48b9cf01	yes (50 min)	9 out of 9	yes	no	-
65884701	no	-	no	no	-
701b7301	-	-	yes	no	-
7262d006	yes (5 min)	4 out of 5	yes	no	-
7fa39f01	no	-	yes	yes	-
b8993403	yes (12 min)	1 out of 1	yes	no	-
badd9e01	no	-	yes	no	-
caea9c01	no	-	yes	no	-
cc366801	yes (60 min)	10 out of 10	no	no	-
d476cd01	no	-	yes	no	-
df9df201	no	-	yes	no	-
e7cd3901	-	-	no	yes	-
f14eb101	yes (53 min)	11 out of 11	no	no	yes
f658d801	yes (2 min)	1 out of 4	yes	no	-

Table 1: The table describes the results of our evaluation. The *Input* column describes whether our approach was able to discover valid input strings. As previously mentioned, some binaries did not have any predefined input strings (hence such binaries are marked by the - in this column). The *Number of Inputs* column describes how many input strings our method discovered out of the total number of input strings defined by the binary.

protocol generation part, our *Pin* tool recorded basic block executed and **not** individual instructions. In the future, we wish to explore hardware performance counters and see if they can also be used for the protocol generation. Another option for finding input strings of a binary is to use the Unix *strings* utility, which finds all null terminated strings inside a binary. One problem is that the *strings* utility may return a number of other strings within the binary, strings not used for input validation, resulting in high false positives. Additionally, *strings* is restricted to only finding static strings, whereas our approach can find dynamically generated input strings. Another option is the use of the *ltrace* tool which echoes library functions called and their arguments. *ltrace* could be used to determine calls of the *strcmp()* function, which can help identify valid input strings. However, a program can be implemented to dynamically detect if it is being traced by a utility such as *ltrace*. We argue that a complex input validation process can make the *ltrace* and *strings* utlitities less efficient than our method. For example, one of the binaries used in the testing of our method, used a tree structure to represent valid input strings (see Appendix A, Example 2). Each node in the tree structure represented a valid character and its depth in the tree represented its index in the input string. Such an input checking method greatly impacts the efficiency of alternative approaches such as the *strings* and the *ltrace* utilities. The *strings* utility would not be useful since the input strings are represented one character at a time. Regardless of whether the binary is designed to detect when it is being traced, using an *ltrace*-based technique would also fail since the program compares characters and does not use a library function (such as *strcmp()*). On the other hand, our approach successfully finds the valid input strings despite the complexity of the input validation mechanism.

In the future, we wish to improve our current method. First, we would like to extend to handle binary input. Furthermore, we would like to apply the input finding technique recursively. It is easy to imagine programs for which the second input varies based on the value of the first input. Also, there obviously exist programs for which some inputs have no predetermined value (such as registering a username). It would be helpful to determine such cases, provide some input and continue the execution of the program, applying the input finding technique once again. However, there are challenges with this, such as determining when to not apply the finding input method because the program has reached an already observed and tested state.

With respect to the protocol state machine, currently our method uses the input finding technique to determine valid arguments. However, the protocol state machine does not currently handle situations in which there are no predefined arguments (e.g. given input *CREATE* the arguments could be predefined such as *dog* or random such as any number). Ultimately, we would like to generate some random arguments that remain constant for every input string in a given permutation. Moreover, our current approach is not capable of identifying equivalent states. As seen in Figure 4, the different authentication paths (use of *HELLO* vs. omission of *HELLO*) lead to different states when in reality the two states (i.e. user is authenticated) are identical. Although we have not yet explored this, state minimization approaches for finite state machines may be applicable for this problem.

Also, our method is based on the assumption that given a number of input strings, the majority of input string permutations are not valid and do not reflect the expected protocol. There may be cases where this assumption does not hold. A better option might be to monitor other aspects of the program, particularly the *.data* and the *.bss* sections and the heap during the executions and see how they react to various input string permutations. Also, we believe hardware performance counters (e.g. branch instructions) may be used to learn more about the program's state and whether a given input caused the program to change states. In the future, we would like to address the above mentioned concerns with our approach for generating protocol state machines.

7. CONCLUSIONS

This paper introduces two novel techniques for autonomously generating valid input strings for closed x86 binaries and for generating a protocol state machine using the discovered input. For generating valid input strings, our method exploits the number of instructions retired to infer whether a given character is valid for a given index of an input string. This process allows us to recursively and incrementally craft valid inputs. To further gain knowledge about the user interaction with the program, we test the program with the inputs generated and construct a protocol state machine which depicts the expected order of the input strings.

We show that our proposed methods are efficient and successful by testing them on 24 DARPA binaries from the Cyber Grand Challenge [5]. Unlike previous works, our methods do not have any fundamental limitations (such as constraint solvers) and are robust enough to handle program binaries of other formats.

We believe that the our techniques could be applied and used in a number of settings including the building of test suites, discovering unwanted or unexpected control flow paths, and finding bugs (when used in combination with a fuzzer).

8. ACKNOWLEDGMENTS

The authors would like to thank Thuan Pham for his feedback and useful discussions.

9. REFERENCES

[1] zzuf: Multi-purpose fuzzer. http://caca.zoy.org/wiki/zzuf.

[2] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 317–329. ACM, 2007.

[3] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[4] P. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex: Protocol specification extraction. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 110–125, May 2009.

[5] DARPA. Cyber grand challenge binaries. https://github.com/CyberGrandChallenge/samples, 2014–2015.

[6] erenyagdiran. I was just asked to crack a program in a job interview ! 2014.

[7] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[8] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[9] A. Lanzi, L. Martignoni, M. Monga, and R. Paleari. A smart fuzzer for x86 executables. In *Software Engineering for Secure Systems, 2007. SESS '07: ICSE Workshops 2007. Third International Workshop on*, pages 7–7, May 2007.

[10] 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. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[11] S. McCamant et al. FuzzBALL vine-based binary symbolic execution. http://bitblaze.cs.berkeley.edu/fuzzball.html, 2014–2015.

[12] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, Dec. 1990.

[13] B. P. Miller, D. Koski, C. Pheow, L. V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of unix utilities and services. 1995.

[14] D. Molnar, X. C. Li, and D. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *USENIX Security Symposium*, pages 67–82, 2009.

[15] D. Song, D. Brumley, et al. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India. http://bitblaze.cs.berkeley.edu/.

[16] V. M. Weaver and S. A. McKee. Can hardware performance counters be trusted? In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 141–150. IEEE, 2008.

APPENDIX

A. INPUT VALIDATION CODE SNIPPETS

The following are two examples of input validation mechanisms from the DARPA Cyber Grand Challenge sample binaries.

A.1 Example 1

In this example, we can see that the program uses the *strcmp()* function to determine if the user input is a valid command. Because of the nature of the *strcmp()* function, when a character at a given index is valid (i.e. pertains to one of the commands), the function will then move to the next character of the input string and compare that with the next character of the command. In our work, we exploit the change in the number of instructions executed caused by this to find valid input strings.

```
while (readuntil(STDIN, buf, sizeof(buf), '\n') > 0)
    {
        char *tok, *input = buf;
        tok = strsep(&input, " ");
        if (tok == NULL)
            break;
        if (strcmp(tok, "CREATE") == 0)
        {
            tok = strsep(&input, " ");
            if (tok == NULL)
                break;
            if (strcmp(tok, "DATABASE") == 0)
                handle_create_database(input);
            else if (strcmp(tok, "TABLE") == 0)
                handle_create_table(input);
            else
                break;
        }
        else if (strcmp(tok, "INSERT") == 0)
        {
            tok = strsep(&input, " ");
            if (tok == NULL)
                break;
            if (strcmp(tok, "INTO") == 0)
                handle_insert_into(input);
        }
        else if (strcmp(tok, "DELETE") == 0)
        {
            tok = strsep(&input, " ");
            if (tok == NULL)
                break;
            if (strcmp(tok, "FROM") == 0)
                handle_delete_from(input);
        }
        else if (strcmp(tok, "SELECT") == 0)
        {
            tok = strsep(&input, " ");
            if (tok == NULL)
                break;
            if (strcmp(tok, "FROM") == 0)
                handle_select_from(input);
        }
        else if (strcmp(tok, "STATS") == 0)
        {
            handle_stats(input);
        }
        else
            fdprintf(STDOUT, "Invalid query!\n");
    }
```

A.2 Example 2

This example comes from a program implementing a Federal Acquisition Regulation (FAR) section lookup service. FAR is a set of rules governing the federal government's purchasing process. Here the system call *receive()* is used to read input directly from the specified file descriptor. The user input is parsed in the *process()* function and the valid input commands are stored in a tree like structure created in the *main()* function. Since the system call is used to read the user input, the number of user-land instructions executed

will not increase as a function of the input size (as it would in the previous example). Our method is able to successfully detect both the valid input strings and the correct expected input size. On the other hand, alternative approaches such as the *strings* utility and the *ltrace* tool would fail. The *strings* utility can only detect null terminated strings, not characters. An *ltrace* based method would also fail since the program compares characters and does not use a library function for this task (such as *strcmp()*).

```
int process(program_state *s) {
    int ret;
    size_t size;
    char *name;
    char *args;
    process_t method;

  while (1) {
        memset(s->input_buf, 0, sizeof(s->input_buf));
        transmit_str(STDOUT, "> ");

        if (0 != receive(0, s->input_buf, sizeof(s->input_buf) - 1, &size)) {
            return -1;
        }

        remove_newline(s->input_buf);

        if (strlen(s->input_buf) == 0)
            continue;

        name = strtok(s->input_buf, " ");
        args = strtok(NULL, "\x00");

        if (strlen(name) >= COMMAND_NAME_SIZE)
            return -1;

        method = get_command(s, name);

        if (method == NULL) {
            transmit_str(STDOUT, "invalid command: ");
            transmit_str(STDOUT, name);
            transmit_str(STDOUT, "\n");
            continue;
        }

        ret = (method)(s, args);

        if (ret == -1) {
            transmit_str(STDOUT, "command failed: ");
            transmit_str(STDOUT, name);
            transmit_str(STDOUT, "\n");
            return -1;
        }

        if (ret == 0) {
            return 0;
        }
    }
}


int main(void) {
  ...
  node_0x40f0.value = 99; /* c */
  node_0x40f0.method = &cmd_ch_sec;
  node_0x40d0.value = 101; /* e */
  node_0x40d0.child = &node_0x40f0;
  node_0x40b0.value = 115; /* s */
  node_0x40b0.child = &node_0x40d0;
  node_0x4090.value = 95; /* _ */
  node_0x4090.child = &node_0x40b0;
  node_0x4070.value = 104; /* h */
  node_0x4070.child = &node_0x4090;
  node_0x4050.value = 115; /* s */
  node_0x4050.method = &cmd_compress;
  node_0x4030.value = 115; /* s */
  node_0x4030.child = &node_0x4050;
  node_0x4010.value = 101; /* e */
  node_0x4010.child = &node_0x4030;
  node_0x3ff0.value = 114; /* r */
  node_0x3ff0.child = &node_0x4010;
  node_0x3fd0.value = 112; /* p */
  node_0x3fd0.child = &node_0x3ff0;
  node_0x3fb0.value = 109; /* m */
  node_0x3fb0.child = &node_0x3fd0;
  node_0x3f90.value = 111; /* o */
  node_0x3f90.sibling = &node_0x4070;
  node_0x3f90.child = &node_0x3fb0;
  node_0x3f70.value = 99; /* c */
  node_0x3f70.method = &cmd_cur_sec;
  node_0x3f50.value = 101; /* e */
  node_0x3f50.child = &node_0x3f70;
  ...
  node_0x3c70.value = 112; /* p */
  node_0x3c70.sibling = &node_0x3cd0;
  node_0x3c70.child = &node_0x3c90;
  node_0x3c50.value = 116; /* t */
  node_0x3c50.method = &cmd_quit;
  node_0x3c30.value = 105; /* i */
  node_0x3c30.child = &node_0x3c50;
  node_0x3c10.value = 117; /* u */
  node_0x3c10.child = &node_0x3c30;
  node_0x3bf0.value = 113; /* q */
  node_0x3bf0.sibling = &node_0x3c70;
  node_0x3bf0.child = &node_0x3c10;
  node_0x3bd0.value = 104; /* h */
  node_0x3bd0.method = &cmd_search;
  node_0x3bb0.value = 99; /* c */
  node_0x3bb0.child = &node_0x3bd0;
  node_0x3b90.value = 114; /* r */
  node_0x3b90.child = &node_0x3bb0;
  node_0x3b70.value = 97; /* a */
  node_0x3b70.child = &node_0x3b90;
  node_0x3b50.value = 101; /* e */
  node_0x3b50.child = &node_0x3b70;
  s.command_list.value = 115; /* s */
  s.command_list.sibling = &node_0x3bf0;
  s.command_list.child = &node_0x3b50;
  ...
}
```