BCE: Extracting Botnet Commands from Bot Executables

Junghee Lim
Thomas Reps

Technical Report #1668

February 2010

BCE: Extracting Botnet Commands from Bot Executables

Junghee Lim
Comp. Sci. Dept., Univ. of Wisconsin
junghee@cs.wisc.edu

Thomas Reps
Comp. Sci. Dept., Univ. of Wisconsin
reps@cs.wisc.edu

***Abstract*—Botnets are a major threat to the security of computer systems and the Internet. An increasing number of individual Internet sites have been compromised by attacks from all across the world to become part of various kinds of malicious botnets. The Internet security research community has made significant efforts to identify botnets, to collect data on their activities, and to develop techniques for detection, mitigation, and disruption. One way of analyzing the behavior of bots is to run the bot executables and observe their actions. For this to be possible, one needs proper input commands that trigger malicious behaviors. However, it is difficult and time-consuming to manually infer botnet commands from binaries. In this paper, we present a tool called BCE for automatically extracting botnet-command information from bot executables.**

Our experiments showed that the new search strategies developed for BCE yielded both substantially higher coverage of the parts of the program relevant to identifying bot commands, as well as lowered run-time.

***Keywords*-botnet analysis; bot-command analysis; directed test generation; control dependence**

I. Introduction

An increasing number of individual Internet sites have been compromised by attacks from all across the world to become part of various kinds of malicious botnets. Botnets seriously undermine computer security and reliability by conducting illegitimate activities, such as performing large-scale distributed denial-of-service attacks; identity theft; sending spam, trojans, and phishing emails; distributing pirated media; and performing click fraud. Moreover, botnets can quickly grow by using worms to attack vulnerable systems. During the time between an announcement of a vulnerability and a patch for the vulnerability, the potential for bot infiltration is particularly high.

The Internet security research community has made significant efforts to identify botnets, to collect data on their activities, and to develop techniques for detection, mitigation, and disruption. Some bots try to avoid detection by using slow-spreading infection techniques. Some use multiple levels of indirection to make it harder to understand the botnet's structure. Some research on botnet detection is based on network-traffic analysis. The work of Stinson and Mitchell [29] is based on observing differences in how bots and benign programs behave in response to data received from the network. It also uses host-based dynamic analysis: performing taint analysis (where data received over the network is considered to be tainted), applying library-call-level taint propagation, and checking for tainted arguments to selected system calls. The methods described in references [16], [17], [20], [22], [30], [33] use flow-based detection by monitoring network traffic (C&C control activities). Temporal/spatial behavior statistics are also used. Such network-based and behavior-based approaches have several drawbacks: the approaches are (i) costly (runtime overhead to monitor network traffic, space overhead for storing packet logs, etc.), (ii) easily evaded, and (iii) not able to recover the structure of a botnet. Some detection techniques [15] rely on well-known bot communication signatures. (A lot of bot code is reused, and thus the commands and authentication mechanisms are widely known.) However, attackers can easily modify the command-and-control language used by their bots to raise the bar for detection and control.

Botnet-Command Extractor (BCE). We have developed a tool called BCE for extracting botnet-command information from bot executables. BCE aims to provide useful information from analysis of bot executables by automatically extracting proper inputs that trigger malicious behavior. Applications of the information recovered include observing and analyzing malicious behaviors, as well as identifying and mitigating botnets.

A typical way to analyze the behavior of a bot is to run the executable and observe its actions. To carry this out, however, one needs proper inputs to trigger malicious behaviors. Some widely known commands are often used for this purpose. However, most attackers change their commands to evade such dynamic analysis. Also, it is a hard problem to obtain such inputs by manually stepping through the executable. BCE automates the extraction of information about botnet commands and the arguments to commands.

The work described in the paper makes the following contributions:

1) BCE automatically extracts botnet-command information from bot executables, without source code or symbol-table/debugging information. The extracted information includes (a) constant command strings that trigger API-level behaviors, (b) relationships, including type relationships, between the input command string and the actual parameters of an API call, and (c) constraints on the actual parameters of an API call.

```
...
else if(strcmp(cmd,‘‘:!p’’)==0) {
   // (1)
}
else if(strcmp(cmd,‘‘:!p2’’)==0) {
   // (2)
}
else if(strcmp(cmd,‘‘:!ppp’’)==0) {
   // (3)
}
```

(a)

```
...
else if(*cmd++ == ‘:’
     && *cmd++ == ‘!’
     && *cmd++ == ‘p’) {
  if(*cmd == 0)
    // (1)
  else if(*cmd == ‘2’)
    // (2)
  else if(*cmd++ == ‘p’
       && *cmd++ == ‘p’)
    // (3)
}
```

(b)

```
procedure foo
. push offset aP1; ‘‘:!p’’
. lea eax, [ebp+arg_0]
. push eax
. call strcmp
. add esp, 0Ch
. or eax, eax
. jnz short loc_402210
. ... // (1)
. push offset aP1; ‘‘:!p2’’
. lea eax, [ebp+arg_0]
. push eax
. call strcmp
. add esp, 0Ch
. or eax, eax
. jnz short loc_402210
. ... // (2)
. push offset aP1; ‘‘:!ppp’’
. lea eax, [ebp+arg_0]
. push eax
. call strcmp
. add esp, 0Ch
. or eax, eax
. jnz short loc_402210
. ... // (3)
```

(c)

Figure 1. (a) A snippet of the EvilBot source code, (b) alternative source code, (c) the assembly code of (a).

The information obtained via BCE can be used to build up proper input commands that trigger API-level behaviors.

2) BCE is able to provide a specification of the API-level behaviors of a bot program without running the bot. Along with the input-command strings extracted from a bot program, BCE also provides a sequence of API calls controlled by each command, which can help the user understand the API-level behavior.

3) BCE is not based on signatures. Some recent approaches to finding out botnet commands are based on pattern-matching techniques. Many bot programs use standard string-library functions to process the input command string, as shown in Fig. 1(a). The assembly code of Fig. 1(a) obtained using the IDAPro disassembler is shown in Fig. 1(c). One can find a pattern in the assembly code: there are two `push` instructions, one of which is for a constant string that IDApro readily identifies, followed by a call to `strcmp`. However, such a technique is ad hoc and can be easily evaded, e.g., by changing the code in Fig. 1(a) to use byte-by-byte comparison instead of using standard library functions, as shown in Fig. 1(b).

4) BCE uses directed test generation [13], enhanced with a new search technique that uses control-dependence information [11] to direct the search. Our experiments show that the method provides higher coverage of the parts of the program relevant to identifying bot commands, as well as lowered run-time.

5) We performed experiments with four real bot programs. Our preliminary results show that BCE is able to effectively extract bot-command information.

Organization. The remainder of the paper is organized as follows: §II discusses what kind of information BCE extracts, and how one can make use of the information to trigger potentially malicious behaviors from a bot. §III presents background on directed test generation [13]. §IV presents the enhanced techniques for exploring program paths that we developed for use in BCE. §V describes the use of nondeterminism in BCE, which is used for writing “harness” code to model possible client environments, possible inputs, and possible return values from library functions or system calls. §VI discusses additional information that BCE recovers, which combines the recovered information about constraints on inputs with type information for the target API calls. §VII describes how a language-independent BCE implementation was created. §VIII presents experimental results. §IX discusses the limitations of BCE. §X discusses related work. §XI concludes.

II. BOTNET-COMMAND EXTRACTOR (BCE)

In this section, we first discuss what information BCE relies on to extract botnet commands. We then summarize the kind of information that BCE provides, and how one can make use of such information to generate proper input commands.

A. *What* BCE *Relies On*

1. API prototypes: BCE relies on information about function prototypes of API functions. For example, the prototype of *ShellExecute* is as follows:

```
HINSTANCE ShellExecute(
    HWND hwnd,
    LPCTSTR lpOperation,
    LPCTSTR lpFile,
    LPCTSTR lpParameters,
    LPCTSTR lpDirectory,
    INT nShowCmd
);
```

`lpDirectory`: *[in] A pointer to a null-terminated string that specifies the default (working) directory for the action.*

The function prototypes are used to construct reasonable input commands given the command specification extracted by BCE.

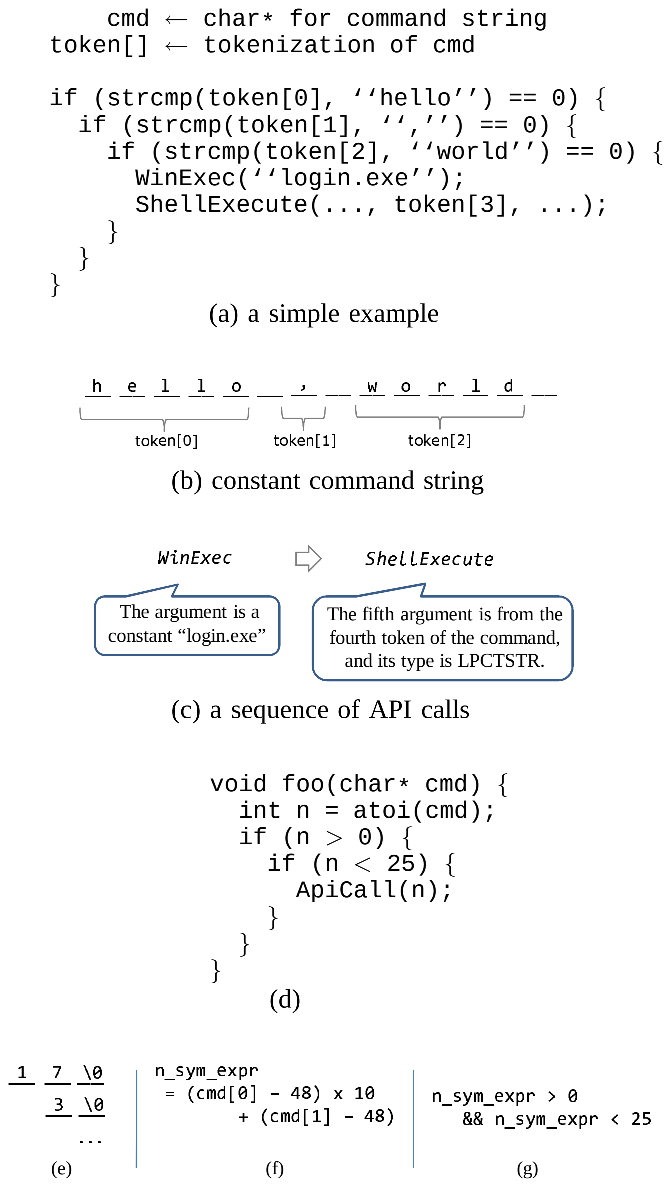
2. Control-Dependence Graph: BCE makes use of the control-dependence graph for a bot binary to optimize its state-space-exploration algorithm. We discuss the use of control dependences in more detail in §IV.

B. *What* BCE *Recovers and How to Use the Recovered Information*

1. Constant command strings that control a bot. For example, there are three nested if-statements in the code shown in Fig. 2(a). Two API calls are invoked when the three branch conditions are satisfied. Suppose that `cmd` has been tokenized into three null-terminated strings. Fig. 2(b) is the command string constructed based on the information extracted by BCE. This information is obtained from conditional branches where a portion of the command string is compared against some constants, as the three strings (“hello”, “,”, and “world”) in the example.

2. A sequence of API calls controlled by each command. Along with each command, BCE provides a sequence of API calls that are controlled by the command. For example, the code executed when the command string shown in Fig. 2(b) is issued subsequently invokes *WinExec* and *ShellExecute*. This information can be directly used to get an idea of the API-level behavior of a bot without actually executing it.

3. Information about the actual arguments of each API call. In addition to a sequence of API calls, BCE provides information about the arguments to each API call, such as constant values for an argument, symbolic expressions, and constraints on the symbolic expressions, as shown in Fig. 2(e), (f), and (g), respectively.



```
   cmd ← char* for command string
token[] ← tokenization of cmd

if (strcmp(token[0], ‘‘hello’’) == 0) {
  if (strcmp(token[1], ‘‘,’’) == 0) {
    if (strcmp(token[2], ‘‘world’’) == 0) {
      WinExec(‘‘login.exe’’);
      ShellExecute(..., token[3], ...);
    }
  }
}
```

(a) a simple example

(b) constant command string

WinExec
ShellExecute
The argument is a constant “login.exe”
The fifth argument is from the fourth token of the command, and its type is LPCTSTR.

(c) a sequence of API calls

```
void foo(char* cmd) {
  int n = atoi(cmd);
  if (n > 0) {
    if (n < 25) {
      ApiCall(n);
    }
  }
}
```

(d)

(e) (f) (g)

Figure 2. (a) A simple example program; (b) the command string constructed based on the information obtained from BCE; (c) a sequence of API calls obtained from BCE; (d) another simple example; (e) constant examples provided by BCE; (f) the symbolic expression obtained from BCE for the argument n; (g) the constraint obtained from BCE.

- *Constant arguments:* In many cases, API calls take constant arguments that one can statically extract from binaries. For example, the first argument of *WinExec* in Fig. 2(a) is a constant string “login.exe”. In addition to the sequence of API calls, information about argument values enables one to get a better idea of the API-level behavior of a bot without running it.
- *Symbolic expressions in the input-state vocabulary:*

BCE also provides a symbolic expression for each actual parameter of an API call, along with its type information, as long as the argument is related to some part of the input command. For example, *ShellExecute* in Fig. 2(a) takes the fourth token of the input command as its fifth argument. BCE automatically extracts a symbolic expression that has one symbolic term, `token[3]`, along with its type LPCTSTR. The type information is obtained from the prototype of the API call. The type information is used to come up with a proper input string. Given the information that the fourth token is supposed to be a null-terminated string that specifies a working directory name, one can build up a complete command string as follows:

```
"hello , world C:\temp"
```

Fig. 2(f) shows another example of a symbolic expression that BCE provides. Fig. 2(f) is the symbolic expression obtained for n in Fig. 2(d). In Fig. 2(d), the input command string is a numeral, which is converted into a number by calling `atoi`; the number is then passed into an API call as an argument. The symbolic expression is in the *input vocabulary* in that the symbols (`cmd[0]` and `cmd[1]`) that appear in it represent individual byte values of the input command string. We discuss how the symbolic expression is generated in §III.

- *Constraints on symbolic expressions:* BCE also provides constraints on the symbolic expressions extracted for each actual parameter of an API call, if any. For example, BCE extracts the constraint shown in Fig. 2(g) for the actual parameter n to the API call in Fig. 2(d).

  This constraint is obtained from the two conditional branches that guard the API call. BCE finds out the conditional branches on which the API call transitively depends. It only collects branches whose predicates constrain the given symbolic expression.

  The obtained constraints also play an important role for building up proper input commands. BCE provides some concrete examples for n, as shown in Fig. 2(e): the numeral strings "17" and "3" satisfy the two branch predicates ($n > 0$ and $n < 25$). Therefore, these input strings cause the API call to be invoked, and thus can be directly used to run the bot program. However, there are cases when the automatically generated concrete examples fail to trigger observable behavior of a bot. For example, suppose the API in Fig. 2(d) is some API that takes an IP address and sets up a connection to the server (e.g., `httpserver` of SpyBot). Because concrete examples are randomly selected to satisfy the constraints collected during symbolic execution, it is not likely that BCE finds out a reasonable IP address unless there are conditional branches where it can extract proper constraints on the command. Therefore, in some cases, the user is responsible for making use of the extracted constraints to construct reasonable inputs.

§VI discusses other kinds of information about the bot's commands that BCE provides—in particular, information that combines the recovered symbolic information about inputs with type information for the target API calls.

III. Overview

This section provides background on *directed test generation* [13], which collects path constraints and uses them to explore new paths systematically. In applying directed test generation in BCE to the problem of extracting bot commands, we developed new techniques to explore program paths, which differ from conventional directed-test-generation techniques. We discuss our enhanced search algorithms in §IV.

One example of a directed test-generation tool is SAGE [14], which is a whitebox fuzz-testing tool, an advance on fuzz testing based on random mutations. SAGE records an actual run of a program under test, starting with a well-formed input, then symbolically evaluates the recorded trace and generates constraints that capture how the program uses its inputs. The generated constraints are then systematically modified and solved with a constraint solver to produce new inputs that cause the program to follow different control-flow paths. The process is repeated with a coverage-maximizing heuristic designed to find defects as fast as possible. Fig. 4 shows a simple example taken from [14]. There are 5 values leading to the error out of 2^{8*4} possible values for 4 bytes. Therefore, the probability of hitting the error with random testing is about $1/2^{32}$. In contrast, whitebox dynamic test generation can find the error in at most $2^4 = 16$ iterations (4 valid path constraints are collected during the exploration process).

Alg. 1 shows the basic search step of the BCE algorithm. The outline of the algorithm is similar to typical directed-test-generation techniques, which can be roughly summarized as repeatedly applying the following three steps:[1]

(1) Concrete execution. Perform concrete execution along some path π. The concrete-execution loop repeatedly takes a concrete program state and returns an updated state that captures the semantics of each instruction. A program state keeps the values of (i) registers, (ii) flags, and (iii) memory locations.

[1]The first step (concrete execution) and the second step (symbolic execution) can be done simultaneously, which is sometimes called *concolic execution* [27]. In concolic execution, concrete values from the concrete execution state are sometimes used to simplify the symbolic states created during symbolic execution.

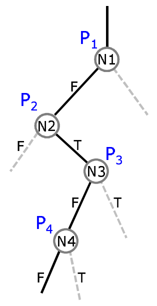


Figure 3. A trace tree. P_1, P_2, P_3, and P_4 denote the branch constraints obtained during the symbolic execution of the path shown in bold.

(2) Symbolic execution. Perform symbolic execution along the same path π. Symbols represent values in the input state and the input command. Symbolic execution is similar to concrete execution except that values may be symbolic terms or formulas over the input symbols. At each branch node B, the branch condition is symbolically evaluated and then constrained to the Boolean value needed to follow the branch direction taken by π at the current instance of B. This yields a branch constraint for B.

For the symbolic memory state, we use a map from concrete scalars (i.e., addresses) to symbolic expressions. In machine code, int-valued quantities and address-valued quantities are indistinguishable at runtime, and arithmetic on addresses is used extensively. Therefore, at a program point, the address for either a memory-access or memory-update operation may be a symbolic expression. The BCE algorithm concretizes such symbolic expressions using the value for the address calculated from the concrete state at the program point.

(3) Exploring a new path. To explore a new path, the algorithm chooses some branch node B in the concrete trace, and conjoins all the branch-constraint formulas encountered on the prefix of π from the root node to to B. In place of the branch-constraint formula φ for B itself, $\neg\varphi$ is used. For example, if one wants to explore the path toward the true branch of N_4 in Fig. 3, the following path-constraint formula is passed to the constraint solver:

$$P_1 \ \wedge \ P_2 \ \wedge \ P_3 \ \wedge \ \neg P_4.$$

The constraint solver determines whether the path-constraint formula is satisfiable, and, if it is satisfiable, returns a satisfying assignment for it, which is used to generate a new concrete input state for use in the next round of exploration.

BCE maintains a trace tree that is expanded during the process of symbolic execution. Each node in a trace tree represents a different execution instance of a branch instruction in the program. Each node can have two children, one of which represents the first branch node encountered along the path through the true successor, the other of which is the first branch node along the path through the false successor. The path from the root node to a leaf node represents the branch instructions of a concrete trace. Each edge holds a branch constraint obtained from symbolic execution. Each time a branch is symbolically executed (to follow the direction taken by a previous concrete execution), the trace tree is extended appropriately.

Algorithm 1 Single BCE Iteration

```
Require: A concrete state $S$.
Require: A trace tree $T$
 1: Concretely execute the program with the concrete state $S$.
 2: Let $CT$ be the concrete trace obtained from the concrete execution.
 3: Symbolically execute the trace $CT$.
 4: Let $T'$ be the trace tree augmented by the symbolic execution.
 5: if at least one API call is encountered in the concrete trace then
 6:     Based on the symbolic state obtained in the symbolic execution, collect information about the command tokens that appear in the arguments to each API call.
 7: end if
 8: repeat
 9:     Choose a new path $\pi$ in the trace tree $T$.
10:     Let $\varphi$ be the path-constraint formula obtained by conjoining the branch constraints along $\pi$.
11: until $\varphi$ is satisfiable
12: Let $M$ be the model obtained by calling the constraint solver with $\varphi$.
13: Create the new concrete state $S'$ updated with the assignments from the model $M$.
```

```
void top(char input[4]) {
  int cnt = 0;
  if (input[0] == 'b') cnt++;
  if (input[1] == 'a') cnt++;
  if (input[2] == 'd') cnt++;
  if (input[3] == '!') cnt++;
  if (cnt >= 4) abort();
}
```

Figure 4. An example for whitebox fuzz testing

IV. Program Exploration using Control-Dependence Information

This section presents the enhanced techniques for exploring program paths that we developed for use in BCE. MineSweeper [5] and the work of Moser et al. [25] have shown the potential for carrying out better exploration in malware. Other tools, such as SAGE, have addressed the problem of path explosion by introducing heuristics to improve coverage [14]. SAGE uses so-called *generational search* designed to partially explore the state spaces of large applications with the aim of finding bugs faster. As in most of other directed-test-generation tools, SAGE aims to improve test coverage. Unlike bug-finding tools or tools that aim to improve coverage, in BCE we are interested in *goal-directed techniques* aimed at extracting bot commands.

The characteristics of how the bot code parses the transmitted commands and takes actions depending on the parsed commands can be used to come up with better exploration strategies that avoid possible explosion and obtain more complete specifications about the command structure. We incorporated the following path-exploration strategies into BCE:

- Choose as a candidate for the new path the branches that have a possibility of leading to API calls.[2]
- Prune the search performed by BCE so that each path includes a limited number of API calls if a candidate branch for extending the path is independent of the branches involved with the API calls already found in the path.

The exploration strategies are based on the fact that our goal is to identify as many feasible input commands as possible that lead to API calls of interest.

To identify branches that have a possibility of encountering API calls, we use *control-dependence information*. §IV-A discusses control-dependence information. In §IV-B and §IV-C, we present how control-dependence information is used in BCE.

A. Control Dependence

The *control dependence* relation is one of the fundamental relationships among statements or instructions used in compilers and optimizers. For instance, control-dependence information is used in compilers to determine whether it is safe to reorder or parallelize statements [11]. A control dependence holds when the decision made at a branch X controls whether another statement or instruction Y is executed.

Control dependence is defined in terms of the post-domination relation.

Definition 4.1: Node Z *post-dominates* node X iff $Z \neq X$ and all paths from X to the end of the procedure include Z. (Note that by this definition a node does not post-dominate itself.)

Definition 4.2: Node Y is *directly control dependent on node X* iff

1) there exists a path π: $X \rightarrow^{+} Y$ such that Y post-dominates every node in π different from X, and
2) X is not post-dominated by Y.

We use C to denote the direct-control-dependence relation.

[2]BCE is parameterized to take a list of interesting API entry points of interest.

```
if (a > 0) { // (b1)
  b = 1; // (s1)
  if (a < 25) { // (b2)
    c = 2; // (s2)
  }
}
else {
  d = 3; // (s3)
}
e = 4; // (s4)
```

Figure 5. An example to show control dependences.

Control dependences can be broken down more finely into dependences on the true branch or false branch of a branch-node X, as follows:

Definition 4.3: Node Y is *directly control-dependent on edge $X \rightarrow W$* iff

1) there exists a path π: $W \rightarrow^{*} Y$ such that Y post-dominates every node in π different from X, and
2) X is not post-dominated by Y.

We say that the relation $C_t(X, Y)$ holds when X is a branch node and Y is directly control dependent on X's true branch. C_f is defined similarly.

Each branch node is associated with two sets of CFG nodes: one consists of the transitive control-dependence successors for its true branch (denoted by C_tC*); the other consists of the transitive control-dependence successors for its false branch (denoted by C_fC*).

$$C_tC* : \text{True control successors}$$
$$C_fC* : \text{False control successors}$$

For example, in Fig. 5, the statements `(s1)` and `(s2)` are transitively control dependent on the true branch of `b1`; statement `(s3)` is transitively control dependent on the false branch of `b1`. Statement `(s4)` is not transitively control dependent on any branch in this example. (Henceforth, we will abbreviate "transitive control dependence" by "control dependence".)

In the next section, we discuss a novel usage of control-dependence information in BCE.

B. Choosing Interesting Branches using Control-Dependence Information

BCE uses control-dependence information (CDI) to annotate the trace tree. If there is at least one API call in C_tC* (or C_fC*) of a branch node, the node is marked as N_t (or N_f). Any branch that has a call to a function that contains at least one N_t or N_f in C_tC* (C_fC*) is also marked as N_t (or N_f). BCE only chooses one of the nodes marked with N_t or N_f as a candidate for the new path. Fig. 6 compares an exploration strategy that uses control-dependence information (CDI) to one that does not. The solid lines in the figures indicate

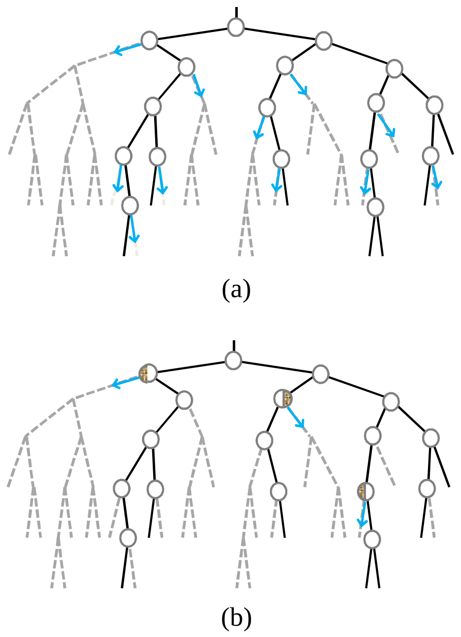


Figure 6. Two trace trees; (a) A trace tree without CDI; (b) a trace tree with CDI; the circles represent branch nodes; the solid arrows represent possible paths to explore; the half-shaded circles represent nodes labeled as either N_f or N_t.

```
[1]  char* p1; // input;
[2]  char p2[] = "bot.execute";
[3]  int v;
[4]  char c1;
[5]  do {
[6]    c1 = *p1++;
[7]    c2 = *p2++;
[8]    v = (unsigned)c1 - (unsigned)c2;
[9]    if(v != 0)
[10]      break;
[11] } while(c1 != '\0');
[12]
[13] if(v == 0)
[14]    APICall
```

Figure 7. An example in which it is necessary to choose an alternative candidate as a new path; the source code of `strcmp` is inlined in this example.

the paths that have previously been explored. One chooses as the next candidate one of the nodes (on the solid lines in Fig. 6) that has a solid edge to only one child. Such choices are marked with arrows. There are fewer candidates to explore in Fig. 6(b) than in Fig. 6(a). The degree of the improvement by using CDI depends on the percentage of nodes marked with N_t or N_f. We discuss how the approach works out with real bot programs in §VIII.

Algorithms. Alg. 2 and Alg. 3 describe the path-exploration algorithm of BCE. In Alg. 2, BCE chooses a node n in the trace tree marked as N_f or N_t whose corresponding branch

Algorithm 2 ChooseNewPath

```
Require: A trace tree $T$
Ensure: Formula $\varphi$
1: Let Frontier be the branch node in $T$ that is either marked as $N_f$ and does not have a false child in $T$, or marked as $N_t$ and does not have a true child in $T$, and has the shortest path from the root node.
2: Let $\varphi$ be the formula conjoined with all the formulas associated with the branches on the path from Frontier back to the root node.
3: Return $\varphi$
```

Algorithm 3 GenerateNewConcreteState

```
Require: A trace tree $T$
Ensure: A concrete state CS$'$
1: $\varphi$ = ChooseNewPath($T$)
2: Call the constraint solver with the formula $\varphi$
3: if $\varphi$ is feasible then
4:    Let $M$ be the model from the constraint solver
5:    Let CS be a random concrete state
6:    Let CS$'$ be CS updated with all the assignments in $M$
7:    Return CS$'$
8: else
9:    Let $T'$ be $T$ augmented with a dummy node at the previously selected node
10:   GenerateNewConcreteState($T'$)
11: end if
```

is not in the trace tree. BCE then conjoins all the formulas of the branches on the path from n back to the root node. Alg. 3 takes that formula and calls a constraint solver to obtain a model. If the formula for the path that BCE chose to explore is feasible, it generates a new concrete state that gets used in the next round of exploration. Otherwise, it augments the trace tree so that the previously explored path is never selected again, and calls itself recursively.

Fig. 8(a) is an example in which the number of possible execution paths is exponential in the number of branches: each of the 5 `if`-statements is independent of each other. For this code fragment, BCE takes 8 iterations when it uses CDI,[3] of Alg. 1 to identify 2 different paths (one toward the API call inside the second `if`-statement, and the other toward the fifth statement) whereas without CDI it exhibits exponential behavior.

Indirect control-dependence. In some cases, it is possible that a candidate node marked as N_t or N_f has a branch predicate, the negation of which causes the path constraint

[3]The body of `strcmp` includes some branches to compare an individual character of the first argument with one constant character from the second argument. To get to the two API call sites, BCE needs several trials for each.

```
[1]  if(strcmp(c[0], "aaa")==0) {
[2]    n = atoi(c[5]);
[3]  }
[4]  if(strcmp(c[1], "bbb")==0) {
[5]    APICall1(...);
[6]  }
[7]  if(strcmp(c[2], "ccc")==0) {
[8]    n = atoi(c[5]);
[9]  }
[10] if(strcmp(c[3], "ddd")==0) {
[11]   n = atoi(c[5]);
[12] }
[13] if(strcmp(c[4], "eee")==0) {
[14]   APICall2(...);
[15] }
```

(a)

```
[1]  if(strcmp(c[0], "aaa")==0) {
[2]    n = atoi(c[5]);
[3]  }
[4]  else if(strcmp(c[1], "bbb")==0) {
[5]    APICall1(...);
[6]  }
[7]  else if(strcmp(c[2], "ccc")==0) {
[8]    n = atoi(c[5]);
[9]  }
[10] else if(strcmp(c[3], "ddd")==0) {
[11]   n = atoi(c[5]);
[12] }
[13] else if(strcmp(c[4], "eee")==0) {
[14]   APICall2(...);
[15] }
```

(b)

Figure 8. (a) An example with independent `if`-statements (and thus an exponential number of paths). (b) An example more typical of bot code (with a linear number of paths).

to be infeasible, that does not help program exploration. For example, in Fig. 7, `p1` points to the input character array, and `p2` points to the constant string `"bot.execute"`. The branch on line [13] is marked as N_t because its true branch contains an API call. Suppose that in the initial concrete state, the first input byte pointed to by `p1` is something different from `'b'`, and thus the loop in lines [5]–[11] terminates at line [9] after one iteration with the condition `v != 0`, and the false branch of line [13] is executed. In the subsequent symbolic execution in which the character array pointed to by `p1` is treated as a list of symbols, the path constraint toward the true branch at line [13] is

$$(S_{c1} - C_b \neq 0) \ \wedge \ (S_{c1} - C_b = 0),$$

where S_{c1} is a symbol that represents the first input byte, and C_b is a constant symbol. This formula is infeasible. In such cases, as a heuristic, BCE chooses branches prior to the candidate node on the trace as an alternative candidate. In this example, the false branch at line [9] is chosen as a new path so that from the path constraint

$$S_{c1} - C_b = 0,$$

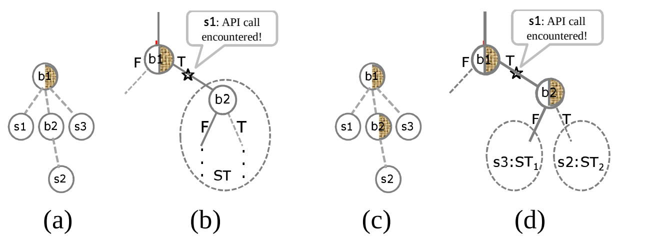


Figure 9. (a) A control-dependence graph; (b) a trace tree when sub-trees are pruned using control-dependence graph (a); (c) another control-dependence graph; (d) the trace tree when sub-trees are pruned using control-dependence graph (c).

the constraint solver can provide a new test input in which the first input byte equals $'b'$.

When a situation occurs like the one described for line [13], a command-line flag controls how many prior branches to try.

C. Pruning the Trace Tree using Control-Dependence Information

CDI helps to direct program exploration toward API call sites. However, even when some candidate branches are excluded by CDI, there is still the possibility of combinatorial explosion. For example, in Fig. 8(a), there are 24 paths in total that invoke the API call(s): there are 8 paths that invoke each call (and not the other) and an additional 8 that invoke both. When the branches controlled by different commands are independent of each other, it means that multiple commands can be combined to produce different sequences of API calls. In other words, if there are n independent `if`-statements involved with API calls, the total number of possible paths that invoke at least one API call is 2^n.

To avoid such combinatorial explosion, we limit the exploration performed by BCE so that each path includes a limited number of API calls if a candidate branch for extending the path is independent of the branches involved with the API calls already found in the path. In particular, the path exploration in BCE only finds n paths when there are n independent `if`-statements involved with API calls. The information obtained in this way is still useful to a user, although it shifts the burden onto the user to identify the API-level behaviors of a bot by trying various combinations of the n extracted commands. For the example in Fig. 8(a), BCE only extracts

“bbb” for the second token of cmd
“eee” for the fifth token of cmd

and the user can try running the bot with the three combinations—“bbb”, “eee”, and “bbb” + “eee”—to observe possibly different behaviors.

The heuristic for avoiding combinatorial explosion is performed by pruning the trace tree dynamically. The following code illustrates what is involved in dynamically pruning the trace tree. Fig. 3(a) is the control-dependence graph of the code, and Fig. 3(b) is the corresponding trace tree.

```
if (strcmp(token[0], ‘‘hello’’) $==0$) { // $b1$
  APICall1(...) // $s1$
  if (atoi(token[1]) $>0$) // $b2$
    ... // $s2$
  ... // $s3$
}
```

An API call is invoked immediately in the true branch of $b1$. In this case, BCE considers pruning the sub-tree ST of the trace tree starting from $b2$. The control-dependence information is used to determine whether the sub-tree ST is to be excluded from further exploration. ST can be excluded if it does not include any node marked as N_t or N_f that is control dependent on node $b2$ (see Fig. 9(b)). If there is at least one other API call in $s2$, as shown in Fig. 9(c) and (d), the true branch remains as a candidate to explore because the second `if`-statement is control dependent on the first one.

In practice, many bot programs are written as shown in Fig. 8(b), where each `if`-statement is dependent on other ones. However, even if when they are rewritten in the form of Fig. 8(a), the pruning technique is effective in practice.

V. OS State

Many formalisms for symbolic analysis of programs support the use of nondeterminism, which is useful for writing “harness code” (code that models the possible client environments from which the code being analyzed might be called), as well as for modeling the possible inputs to a program. A common approach is to provide a primitive that returns an arbitrary value of a given type. Examples include the `SdvMakeChoice` primitive of SLAM [3] and the `havoc(x)` primitive of BoogiePL [4].

In some cases, a value returned from a system call or a Windows-API call is used in a branch condition, as shown in the following example. If *GetCurrentDirectory* returns a value greater than 0, `APICall1` is invoked; otherwise, `APICall2` is invoked.

```
for ($i=0$; $i<3$; $i++$) {
  int $n$ = GetCurrentDirectory(...);
  if ($n>0$) {
    APICall1(...)
  }
  else {
    APICall2(...)
  }
}
```

In the current version of BCE, concrete execution and symbolic execution do not go into system calls and Windows API functions. Instead, BCE keeps a sequence of random numbers (*RandSeq*) for concrete execution, and a sequence of symbols ($\overline{\textit{RandSeq}}$) for symbolic execution. During concrete execution and symbolic execution, the successive values in *RandSeq* and $\overline{\textit{RandSeq}}$, respectively, are used as the successive return values from API call sites. In the above example, there are three calls to *GetCurrentDirectory* in a trace because the loop is executed three times. Each of the three return values comes from successive elements of *RandSeq* and $\overline{\textit{RandSeq}}$. In this way, we model the state of the operating system. Network inputs are modeled similarly.

Algorithm 4 ExtractTypeInformation

```
Require: A function prototype $T$
Require: A symbolic state $S$
Require: The current stack address sp
Ensure: Updated database
1: Let $N$ be the number of arguments of function type $T$
2: for $i=0$ to $N-1$ do
3:     Let $T_i$ be the type of the $i^{th}$ argument of function type $T$
4:     $addr_i = sp + i * param\_size$
5:     CollectTypeInformation($T_i$, $addr_i$)
6: end for
```

VI. Extracting Type Information

§II briefly discussed how one can use the information extracted from BCE to understand a bot program and construct proper input commands. This section discusses some additional information that BCE provides to help users understand the recovered information about the botnet’s commands, based on combining the recovered symbolic information about inputs with type information for the target API calls.

Some extracted constant command strings can be directly used to trigger interesting API-level behaviors of a bot program in cases where there are no additional arguments to a command. However, some of the information extracted about a command is in the form of *symbolic expressions*. A symbolic expression captures the semantics of all the instructions on a specific path from the starting point to the API call site. In some cases, the extracted symbolic expression simply represents a sub-string of the command, whereas there are other cases when the command is converted to another form. A typical action is to convert part of the input string, using the standard library function *atoi*, into a number that is passed to the API call. In other words, the input string holds numerals, whereas the API call receives a number.

Once BCE extracts a symbolic expression for an argument to an API call, it is the user’s responsibility to choose a proper input with which to run the bot based on the symbolic expression. To help in this step, BCE extracts type information for each symbolic expression using the algorithms shown in Alg. 4 and Alg. 5.

Algorithm 5 CollectTypeInformation

```
Require: A type $T$
Require: An address addr
Require: A symbolic state $S$
Ensure: Updated database
 1: if $T$ is a pointer type $T'*$ then
 2:     Let sym_expr be the symbolic expression obtained by looking up addr in $S$.
 3:     Insert the mapping (sym_expr, $T'*$) into the database
 4:     Let addr' be the symbolic expression at address sym_expr in $S$
 5:     if addr' is a scalar then
 6:         CollectTypeInformation($T'$, addr')
 7:     end if
 8: else if $T$ is a basetype then
 9:     Let sym_expr be the symbolic expression obtained by looking up addr in $S$.
10:     Insert the mapping (sym_expr, $T$) into the database
11: else if $T$ is a structure type then
12:     for all $T_i$ a field type of $T$ do
13:         CollectTypeInformation($T_i$, addr + $offset_i$)
14:     end for
15: end if
```

Alg. 4 and Alg. 5 are pseudo-code for collecting type information for each extracted symbolic expression. Our approach uses information about the function prototypes of API calls, as well as a database of OS and network-related types. For example, Fig. 10(a) shows the prototype of `getaddrinfo` and the `struct` types `ADDRINFO` and `sockaddr_in`. `ADDRINFO` is the type of the third and fourth arguments of `getaddrinfo`, and `sockaddr_in` is the type of one of the fields of `ADDRINFO`.

For each API call site, BCE collects type information by calling *ExtractTypeInformation* (Alg. 4). Along with such information, *ExtractTypeInformation* takes the symbolic state at the API call site, and the symbolic expression that represents the current stack pointer. For example, Fig. 10(b) is an example that includes a call to the system call `getaddrinfo`. The first token of the command is converted to a numeric value through `atoi` to be used as `sin_zero` for the `sockaddr_in` object, and the second token is used as `ai_canonname` for the `ADDRINFO` object. BCE calls *CollectTypeInformation* with the actual arguments—`ADDRINFO*` and the current stack pointer—for the third argument of `getaddrinfo`.

For each argument to the API call, it calculates the address of the corresponding stack location, and passes it to *CollectTypeInformation* (Alg. 5), along with the argument type from the function prototype and the symbolic state. *CollectTypeInformation* is a recursive function that tries to associate each type with the corresponding symbolic expression on the stack. Depending on the type, the actions

```
int getaddrinfo (
  char*         nodename;
  char*         servname;
  ADDRINFO*     hints;
  ADDRINFO*     res;
};

struct {
  ....
  char*         ai_canonname;
  sockaddr_in*  ai_addr;
  ....
} ADDRINFO;

struct {
  ....
  unsigned long  sin_zero;
} sockaddr_in;
```
(a)

```
sockaddr_in* s = ...; // malloc
s->sin_zero = atoi(cmd_token[0]);
ADDRINFO* a = ...; // malloc
a->ai_canonname = ...; // malloc
strcpy(a->ai_canonname,
              cmd_token[1]);
a->ai_addr = s;
getaddrinfo(..., ..., a, ...);
```
(b)

Figure 10. (a) The prototypes of `getaddrinfo`, `ADDRINFO`, and `sockaddr_in`; (b) an example code fragment.

are slightly different:

- In the case of a pointer type $T*$, BCE first adds the mapping (sym_expr, $T*$) to the database, and looks up the corresponding value in the symbolic state, and recursively calls *CollectTypeInformation*, passing the value along with the type T of the object referred to. For example, *CollectTypeInformation*(`ADDRINFO*`, sp) recursively calls
  CollectTypeInformation(`ADDRINFO`, $S(sp)$[4])
- In the case of a basetype T, BCE looks up the corresponding value (sym_expr) in the symbolic state, and it adds the mapping (sym_expr, T). For example, the first token of the command is used for the field `sin_zero` of `sockaddr_in` in Fig. 10, which is of base-type `unsigned long`. In this case, BCE collects the information that the associated symbolic expression is of type `unsigned long`.
- In the case of a structure type, such as `struct` or `class`, BCE iterates over the structure's fields, calling *CollectTypeInformation* with each type and the address of the corresponding field. For example, *CollectType-Information*(`ADDRINFO`, $S(sp)$) recursively calls *Col-*

[4] $S(sp)$ denotes a lookup of sp in symbolic state S.

lectTypeInformation(`char*`, $S(sp)$ + offset$_1$), *CollectTypeInformation*(`sockaddr_in*`, $S(sp)$ + offset$_2$), and so forth, where offset$_i$ is the corresponding offset for each field.

VII. Implementation

The BCE implementation has been structured so that it can be retargeted to different languages easily. The core components of the system are language-independent in two different dimensions:

1) The BCE driver implements Alg. 1. It is structured so that one only needs to provide an implementation of concrete execution and symbolic execution of a language. Consequently, this component of the system can be used for source-level languages or for machine-code languages.
2) For machine-code languages, we have used two tools that *generate* the required implementations of concrete execution and symbolic execution from descriptions of the syntax and semantics of an instruction set of interest.

The abstract syntax and concrete semantics of an instruction set are specified using a language called TSL (**T**ransformer **S**pecification **L**anguage) [23]. Decoding (i.e., translation of binary-encoded instructions to abstract syntax trees) is specified using a tool called ISAL (**I**nstruction **S**et **A**rchitecture **L**anguage).[5] The relationship between ISAL and TSL is similar to the relationship between Flex and Bison. With Flex and Bison, a Flex-generated lexer passes tokens to a Bison-generated parser. In our case, the TSL-defined abstract syntax serves as the formalism for communicating values—namely, instructions' abstract syntax trees—between the two tools.

Compared with other specification languages for instruction sets, TSL has one unique feature: from a *single* specification of the concrete semantics of an instruction set a *multiplicity* of static-analysis, dynamic-analysis, and symbolic-analysis components can be *generated automatically*. The TSL system consists of two parts:

- The TSL language for specifying an instruction set's abstract syntax and concrete semantics. TSL is a strongly typed, first-order functional language with a datatype-definition mechanism for defining recursive datatypes, plus deconstruction by means of pattern matching.
- The TSL compiler, which translates a specification to a common intermediate representation (CIR). The CIR generated for a given TSL specification is a C++ template that can be used to create multiple analysis components by instantiating the template in different ways.

[5] ISAL also handles other kinds of concrete syntactic issues, including (a) *encoding* (abstract syntax trees to binary-encoded instructions), (b) *parsing assembly* (assembly code to abstract syntax trees), and (c) *assembly pretty-printing* (abstract syntax trees to assembly code).

TSL has two classes of users: (1) instruction-set specifiers, and (2) analysis developers. The former use the TSL language to specify the concrete semantics of different instruction sets; the latter create new analyses by instantiating the CIR in different ways.

Specifying an Instruction Set. Much of what an instruction-set specifier writes in a TSL specification is similar to writing an interpreter for an instruction set in first-order ML [18]. One specifies (i) the abstract syntax of the instruction set, by defining the constructors for a (reserved, but user-defined) type *instruction*; (ii) a type for concrete states, by defining—e.g., for 32-bit Intel x86—the type *state* as a triple of maps:

$$\textit{state} : \textit{State}(\textit{INT32} \rightarrow \textit{INT8},\ \textit{reg32} \rightarrow \textit{INT32},\ \textit{flag} \rightarrow \textit{BOOL});$$

where *INT32* and *INT8* refer to 32-bit and 8-bit integers, respectively, and *reg32* and *flag* refer to a type for the names of 32-bit registers and a type for the names of condition-codes, respectively; and (iii) the concrete semantics of each instruction by writing a TSL function

$$\textit{state}\ \textit{interpInstr}(\textit{instruction}\ I, \textit{state}\ S)\ \{\ \ldots\ \};$$

Semantic Reinterpretation. Each analysis is defined by reinterpreting the constructs of the TSL meta-language. TSL's meta-language supports a fixed set of base-types; a fixed set of arithmetic, bitwise, relational, and logical operators; and a facility for defining map-types. An analysis developer defines a new analysis component by (i) redefining (in C++) the TSL base-types (*INT32*, *INT8*, *BOOL*, etc.), and (ii) redefining (in C++) the primitive operations on base-types ($+_{INT32}$, $+_{INT8}$, etc.). These are used to instantiate the CIR template. This implicitly defines an alternative interpretation of each expression and function in an instruction-set's concrete semantics (including *interpInstr*), and thereby yields an alternative semantics for an instruction set from its concrete semantics.

For BCE, TSL is used to create several useful reinterpretations of an instruction set:

- By instantiating the CIR with a reinterpretation that performs the standard interpretation (in C++) of the TSL operators, we obtain the instruction interpreter for concrete execution.
- By instantiating the CIR with a reinterpretation that translates operations to the input language of an SMT solver, we obtain a semantics suitable for symbolic execution. (In our implementation, we used the Yices input language [10].)

Control-Dependence Information. The control-dependence information used for the systematic path-exploration of BCE is collected from the control-dependence graph for a bot program. BCE uses CodeSurfer/x86 [2] to obtain the control-dependence graph for a bot program.

API Call Prototypes. BCE uses IDApro [19] and its Fast Library Identification and Recognition Technology (FLIRT)

[12] to identify calls to library functions. It then uses a database of function prototypes and OS and network-related types to extract type information from the recovered symbolic information, as described in §VI.

Library Functions. In BCE, each library-function call is replaced with a simplified model on which concrete and symbolic execution are performed as with other user functions.

VIII. Experiments

We performed experiments on four bot programs. The bots are from different families, and they have different sets of commands. Fig. 11 summarizes the experimental results. The table first shows the size of each program in terms of the number of instructions, and the percentage of the branches marked as N_f or N_t for each program.

The four column listed under "Results" shows the number of traces ending with at least one API call, the total number of iterations performed by BCE,[6] and the number of the command strings that expect one or more arguments. BCE provides a symbolic expression for such arguments, as discussed in §VI.

For dBot and AgoBot, we had source code and we were able to compare the extracted commands with the commands that one can obtain from the source code. In case of AgoBot, there are two commands—"bot.quit" and "bot.die"—that are not counted in the trace numbers, but are actually commands. This is because they are not involved with any Windows API call. Those commands modify some values to change the state of the bot. Even though BCE was able to identify those strings, BCE did not mark them as commands because BCE requires some API call to be controlled by an input string for the string to be classified as a command. Each complete command string, such as "bot.die\0", is extracted through multiple BCE iterations as follows:

"bot.d"
"bot.di"
"bot.die"
"bot.die\0"

If there is no indication that the extracted string is a command (i.e., it controls no API calls), such as "bot.die", there needs to be some manual interpretation of BCE's results, such as whether one should consider an array of bytes in the input that ends with a delimiter (e.g., \0 in case of `strcmp`) to be a command.

We also performed an experiment to determine how well the two state-space-exploration strategies that we introduced in §IV-B and §IV-C perform: one strategy chooses a path that has the possibility of encountering API calls (denoted as "w/ CDI"); the other stops further exploration along the current path once the trace encounters an API call (denoted as "w/ Pruning").

The results are shown in Fig. 12. We compared the number of traces ending with API calls and the total number of iterations under the configuration "w/ CDI" and "w/ Pruning" with three other configurations—(i) "w/o CDI" and "w/ Pruning", (ii) "w/ CDI" and "w/o Pruning", and (iii) "w/o CDI" and "w/o Pruning". BCE performs best using the configuration "w/ CDI" and "w/ Pruning".

One other way in which the four configurations differed is in their ability to report whether all commands had been found. Only the configuration "w/ CDI" and "w/ Pruning" is able to do this; i.e., it exhausted its (pruned) search space and hence could report that there was nothing more to be found. With the other configurations, BCE did not finish even if it had identified all the commands.

As explained in §IV-C, the user must bear in mind that the commands identified are really command fragments, and various combinations of the command fragments must be tried.

IX. Limitations

BCE currently has the following limitations:

1. *Plain (unpacked) binaries.* BCE only handles unpacked binaries. In principle, directed test generation is applicable even for packed binaries by invoking a decoder on the fly during concrete execution. However, the current implementation of BCE need a preprocessing step to obtain control-dependence information, which our implementation obtains from a pre-built control-flow graph. One would need some heuristics other than control-dependence information as an alternative for avoiding combinatorial explosion.
2. *Manual identification of the right starting point.* BCE starts its exploration from some command-processing function other than main. This allows relatively short traces for both concrete execution and symbolic execution, resulting in better overall performance of BCE. Typically, there is some initialization code between the beginning of the main function and the command-processing function that is not relevant to extracting input commands. However, this can be problematic if the initialization code affects concrete execution in significant ways. Finding a way to start BCE from the very beginning of a program with low cost is left for future work.
3. *Approximation.* BCE currently approximates some library function calls by using some simplified models. For example, dBot uses `snprintf` as follows to generate a string in a specific format for the purpose of sending a log to the bot-master.

[6]An *iteration* means one run of the basic search step of the BCE algorithm (Alg. 1); on each iteration, a new path is found that leads to a new concrete state.

Bot Program			Results				Time						
Name	# Instrs.	% Nf/Nt	# Traces	# SymExprs	# Iterations	Trace Leng	Avg.CE	Total.CE	Avg.SE	Total.SE	Avg.PE	Total.PE	Total
dBot	32168	19%	18	7	89	1893	2.6	231.4	4.8	427.3	0.9	831.3	1489.9
AgoBot	54641	36%	17	8	123	4167	7.9	979.1	12.5	1538.7	16.8	2067.6	4585.4
SpyBot	8360	40%	31	10	279	1290	3.9	1074.2	7.2	2003.2	8.5	2374.3	5451.7
EvilBot	2917	29%	17	4	133	2476	2.5	333.8	4.4	589.2	2.5	328.5	1251.5

Figure 11. BCE experiments. The columns, in order, are: the number of instructions (#Instrs); the percentage of nodes marked as either N_f or N_t; the number of unique traces ending with at least one API call; the number of commands for which BCE provides symbolic expressions; the total number of iterations to identify the traces; the average trace length; the average time taken for concrete execution; the total time taken for concrete execution; the average time taken for symbolic execution; the total time taken for symbolic execution; the average time taken for path exploration; the total time taken for path exploration; and the total time taken in seconds. The experiments were run on a Intel P4 1.79GHz machine with 1.49GB RAM.

Bot Program	Configuration			
Name	w/ CDI & w/ Pruning	w/o CDI & w/ Pruning	w/ CDI & w/o Pruning	w/o CDI & w/o Pruning
dBot	18/89 (20%)	18/101+ (<18%)	18/99+ (<18%)	11/142+ (<8%)
AgoBot	17/123 (14%)	17/172+ (<10%)	17/158+ (<11%)	13/167+ (<8%)
SpyBot	31/279 (11%)	28/281+ (<10%)	27/420+ (<6%)	25/528+ (<5%)
EvilBot	17/133 (13%)	14/206+ (<7%)	17/163+ (<10%)	11/308+ (<4%)

Figure 12. BCE experiments. The table reports results for four configurations of BCE: (1) "w/ CDI" and "w/ Pruning", (2) "w/o CDI" and "w/ Pruning", (3) "w/ CDI" and "w/o Pruning", and (4) "w/o CDI" and "w/o Pruning". The numbers reported in each column are the number of unique traces ending with API call(s), the total number of iterations, and the percentage of iterations that resulted in a trace ending with API calls. The experiments were run on a Intel P4 1.79GHz machine with 1.49GB RAM; the symbol "+" after the number of iterations means that BCE with the configuration did not finish (i.e., program exploration could continue infinitely even if all possible commands had been identified.)

```
snprintf(buf, sizeof(buf), ''%s %s\r\n'',
               ..., a[x+1]);
```

where `a[x+1]` is one of the command tokens.

A portion of the command is copied into `buf` in `snprintf`. The `buf` is then passed as a parameter to an API call.

If concrete execution and symbolic execution go inside of `snprintf`, BCE can obtain a symbolic expression for `buf` that contains symbols from the input command. Instead of doing that, to simplify BCE's handling of calls to `snprintf`, we model `snprintf` as a copy operator so that the input command symbol `a[x+1]` is copied into the buffer `buf` ignoring the format string.

4. *Obfuscation on branch conditions.* BCE relies on branch conditions to explore a program. Therefore, if the branch conditions are obfuscated by encryption, it prevents BCE from exploring program paths correctly. For example, fragment (a) below is a normal branch condition that checks a byte value against a constant. As proposed by Sharif et al. [28], the code can be obfuscated as shown in fragment (b). Because it is difficult to invert the hash function, it is infeasible to find `c` given $\mathtt{H}_c$.

```
if (X == c) {
    B
}
```

(a)

```
if (Hash(X) == Hc) {
    run Decrypt(B_E, c)
}
// where Hc = Hash(c), B_E = Encrypt(B, c)
```

(b)

X. Related Work

Dynamic Techniques. J. Caballero et al. proposed techniques that can be used to extract the format of the protocol messages sent from a bot-master by analyzing bot binaries [6]. They introduced a technique called *buffer deconstruction* that builds the message field tree of a sent message by analyzing how the output buffer is constructed. Furthermore, they used type-inference-based techniques to find out the type information of each field of the extracted structure by monitoring how the received (or sent) data is used at places where the types are known, such as system calls. Their technique focuses on extracting message formats given proper inputs that trigger malicious actions, whereas BCE aims to extract such proper inputs.

Machine-Code Analyzers Targeted at Finding Vulnerabilities. A substantial amount of work exists on techniques to detect security vulnerabilities by analyzing source code (for a variety of languages) [31], [24], [32]. Less work exists on vulnerability detection for machine code. Kruegel et al. [21] developed a system for automating mimicry attacks. They used symbolic execution of x86 machine code to discover attacks that can give up and regain execution control by modifying the contents of the data, heap, or stack so that the application is forced to return control to injected attack code at some point after the execution of a system call. Cova et al. [9] used that platform to detect security vulnerabilities in x86 binaries via symbolic execution.

MineSweeper [5], the work of Moser et al. [25], and SAGE have been already discussed in §IV.

Machine-Code Model Checkers. AIR ("Assembly Iterative Refinement") [8] is a model checker for PowerPC. AIR decompiles an assembly program to C, and then checks if the resulting C program satisfies the desired property by applying COPPER [7], a predicate-abstraction-based model checker for C source code.

[MC]SQUARE [26] is a model checker for microcontroller assembly code. It uses explicit-state model-checking techniques (combined with a degree of abstraction) to check CTL properties.

Balakrishnan et al. developed two machine-code model checkers, CodeSurfer/x86 [2] and DDA/x86 [1]. Neither system uses symbolic execution; they are based on abstract-interpretation methods (e.g., numeric abstract domains and a form of abstraction refinement). BCE makes use of CodeSurfer/x86 to obtain the control-dependence graph for a bot program; however, for efficiency reasons, CodeSurfer/x86 was run in a mode in which many of its advanced analysis features were turned off.

XI. CONCLUSION

We developed a tool called BCE that automatically extracts botnet-command information from bot executables, without using source code or symbol-table/debugging information. The information obtained using BCE can be used to build up proper input commands that trigger API-level behaviors. BCE furnishes other kinds of information about a bot's commands, in particular, information that combines the recovered symbolic information about inputs with type information for the target API calls. BCE also provides a sequence of API calls controlled by each command, which helps users to understand the bot's API-level behaviors.

BCE performs directed test generation on executables and incorporates a new search technique based on control-dependence information. Our experiments showed that the new search strategies developed for BCE yielded both substantially higher coverage of the parts of the program relevant to identifying bot commands, as well as lowered run-time.

Acknowledgements

This research was supported by NSF under grants CCF-0540955, CCF-0810053, and CCF-0904371, by ONR under grants N00014-09-1-0510 and N00014-09-1-0776, by ARL under grant W911NF-09-1-0413, and by AFRL under grants FA8750-06-C-0249 and FA9550-09-1-0279. Junghee Lim was supported by a Symantec Research Labs Graduate Fellowship.

REFERENCES

[1] G. Balakrishnan and T. Reps. Analyzing stripped device-driver executables. In *TACAS*, 2008.

[2] G. Balakrishnan, T. Reps, N. Kidd, A. Lal, J. Lim, D. Melski, R. Gruian, S. Yong, C.-H. Chen, and T. Teitelbaum. Model checking x86 executables with CodeSurfer/x86 and WPDS++. In *CAV*, 2005.

[3] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, New York, NY, 2001. ACM Press.

[4] M. Barnett, B.-Y. Chang, R. DeLine, B. Jacobs, and K. Leino. Boogie: A modular reusable verifier for object-oriented programs. 2005.

[5] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*. Springer, 2008.

[6] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Bidirectional protocol reverse engineering: Message format extraction and field semantics inference. Tech. rep. 2009-57, EECS, UC-Berkeley, 2009.

[7] S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent C programs. *Formal Methods in System Design*, 25(2–3), 2004.

[8] S. Chaki and J. Ivers. Software model checking without source code. In *Proc. of the First NASA Formal Methods Symposium*, 2009.

[9] M. Cova, V. Felmetsger, G. Banks, and G. Vigna. Static detection of vulnerabilities in x86 executables. In *ACSAC*, 2006.

[10] B. Dutertre and L. de Moura. Yices: An SMT solver, 2006. http://yices.csl.sri.com/.

[11] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *Trans. on Prog. Lang. and Syst.*, 3(9):319–349, 1987.

[12] Fast Library Identification and Recognition Technology, DataRescue sa/nv, Liège, Belgium, http://www.datarescue.com/idabase/flirt.htm.

[13] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, 2005.

[14] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[15] J. Goebel. Rishi: Identify bot contaminated hosts by irc nickname evaluation. In *USENIX Sec. Symp.*, 2007.

[16] G. Gu, R. Perdisci, J. Zhang, and W. Lee. Botminer: Clustering analysis of network traffic for protocol- and structure-independent botnet detection. In *USENIX Sec. Symp.* 2008.

[17] G. Gu, J. Zhang, and W. Lee. Botsniffer: Detecting botnet command and control channels in network traffic. In *Network and Dist. Syst. Security*. 2008.

[18] E. Harcourt, J. Mauney, and T. Cook. Functional specification and simulation of instruction set architectures. In *Proc. Int. Conf. on Sim. and Hardw. Desc. Langs.* SCS Press, 1994.

[19] IDAPro disassembler, http://www.datarescue.com/idabase/.

[20] A. Karasaridis, B. Rexroad, and D. Hoeflin. Wide-scale botnet detection and characterization. In *HotBots'07: Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*. USENIX Association, 2007.

[21] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *USENIX Sec. Symp.*, 2005.

[22] Y. Kugisaki, Y. Kasahara, Y. Hori, and K. Sakurai. Bot detection based on traffic analysis. In *Intelligent Pervasive Computing*, 2007.

[23] J. Lim and T. Reps. A system for generating static analyzers for machine instructions. In *CC*, 2008.

[24] B. Livshits and M. Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Sec. Symp.*, 2005.

[25] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy*, 2007.

[26] B. Schlich. *Model Checking of Software for Microcontrollers*. PhD thesis, RWTH Aachen University, Germany, 2008.

[27] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *FSE*, 2005.

[28] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *Network and Dist. Syst. Security*. 2008.

[29] E. Stinson and J. C. Mitchell. Characterizing bots' remote control behavior. In *In Lecture Notes in Computer Science*, page 4579. Springer, 2007.

[30] W. T. Strayer, D. Lapsely, R. Walsh, and C. Livadas. *Botnet Detection Based on Network Behavior*. Springer US, 2008.

[31] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, 2000.

[32] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Sec. Symp.*, 2006.

[33] J. Zhuge, T. Holz, XinhuiHan, J. Guo, and W. Zou. Characterizing the IRC-based botnet phenomonon. Tech. rep., Reihe Informatik, 2007.