Towards Generating High Coverage Vulnerability-based Signatures with Protocol-level Constraint-guided Exploration

Juan Caballero, Zhenkai Liang, Pongsin Poosankam, Dawn Song

June 23, 2008
CMU-CyLab-08-009

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

Towards Generating High Coverage Vulnerability-based Signatures with Protocol-level Constraint-guided Exploration

Juan Caballero*, Zhenkai Liang*, Pongsin Poosankam*, Dawn Song*‡
*Carnegie Mellon University ‡UC Berkeley

ABSTRACT

Signature-based input filtering is an important and widely deployed defense. But current signature generation methods have limited coverage and the generated signatures can be easily evaded by an attacker with small variations of the exploit message. In this paper, we propose *protocol-level constraint-guided exploration*, a new approach towards generating high coverage vulnerability-based signatures. In particular, our approach generates high coverage, yet compact, *vulnerability point reachability predicates*, which capture many paths to the vulnerability point. We have implemented Endeavour, a system that implements our approach. Our results show that our signatures have high coverage (optimal or close to optimal in our experiments) and are small (often human-readable), offering dramatic improvements over previous approaches.

1. Introduction

Automatic signature generation remains an important open problem. According to Symantec's latest Internet Security Threat Report hundreds of new security-critical vulnerabilities were discovered in the second half of 2007 [30]. For many of these vulnerabilities, the exploit development time is less than a day, while the patch development time is often days or months [30]. In addition, the patch deployment time can be long due to extensive testing cycles.

To address these issues, *signature-based input filtering* has been widely deployed in IPS and IDSes. Signature-based input filtering matches program inputs against a set of signatures and flags matched inputs as attacks. It provides an important means to protect vulnerable hosts when patches are not yet available or have not yet been applied. Furthermore, for legacy systems where patches are no longer provided by the vendor, or critical systems where any changes to the code might require a lengthy re-certification process, signature-based input filtering is often the only practical solution to protect the vulnerable program.

The key technical challenge to effective signature-based defense is to automatically and quickly generate signatures that have both low false positives and low false negatives. In addition, such signatures should be generated without access to the source code. This is crucial to wide deployment since it enables users and third-parties to generate signatures for commercial-off-the-shelf (COTS) programs, without relying on software vendors, thus enabling a quick response to newly found vulnerabilities.

Coverage is the main open challenge. Due to the importance of the problem, many different approaches for automatic signature generation have been proposed. Early work proposed to generate *exploit-based signatures* using patterns that appeared in the observed exploits, but such signatures can have high false positive and negative rates [20–24, 27, 29, 31, 33, 34]. More recently, researchers proposed to generate *vulnerability-based signatures*, which are generated by monitoring the program execution and analyzing the actual conditions needed to exploit the vulnerability and can guarantee a zero false positive rate [10, 15].

A vulnerability is a point in a program where execution might "go wrong". We call this point the *vulnerability point*. A vulnerability is only exploited when a certain condition, the *vulnerability condition*, holds on the program state when the vulnerability point is reached. Thus, to exploit a vulnerability, the input needs to satisfy two conditions: (1) it needs to lead the program execution to reach the vulnerability point; (2) the program state needs to satisfy the vulnerability condition at the vulnerability point. We call the condition that denotes whether an input message will make the program execution reach the vulnerability point the *vulnerability point reachability predicate*. Thus, the problem of automatically generating a vulnerability-based signature can be decomposed into two: identifying the vulnerability condition and identifying the vulnerability point reachability predicate. A vulnerability-based signature is simply the conjunction of the two. As we will describe later, one of the key challenges is how to generate vulnerability point reachability predicates with high coverage and compact size, which will be the focus of this paper. And we consider the problem of identifying the vulnerability condition as an orthogonal problem and out of the scope of this paper.

The main problem with early vulnerability-based signature generation approaches [10, 15] is that they only capture a single path to the vulnerability point (i.e., their vulnerability point reachability predicate contains only one path). However, the number of paths leading to the vulnerability point can be very large, sometimes infinite. Thus, such signatures are easy to evade by an attacker with small modifications of the original exploit message, such as changing the size of variable-length fields, changing the relative ordering of the fields (e.g., HTTP), or changing field values that drive the program through a different path to the vulnerability point.

Acknowledging the importance of enhancing the coverage of vulnerability-based signatures, recent work tries to incorporate multiple paths into the vulnerability point reachability predicate either by static analysis [11], or by black-box probing [14, 16]. However, due to the challenge of precise static analysis on binaries, the vulnerability point reachability predicates generated using static analysis are too big [11]. And, black-box probing techniques [14, 16], which perturb the original exploit input using heuristics such as duplicating or removing parts of the input message or sampling certain field values to try to discover new paths leading to the vulnerability point, are highly inefficient and limited in its exploration. Hence, they generate vulnerability point reachability predicates with low coverage.

Thus, a key open problem for generating accurate and efficient signatures is how to generate vulnerability point reachability predicates with high coverage and in a compact form. In addition, high coverage vulnerability point reachability predicates have fur-

ther applications such as patch testing. For example, the Microsoft patch MS05-018 missed some paths to the vulnerability point and as a result left the vulnerability still exploitable after the patch [1]. Automatically identifying vulnerability point reachability predicates with high coverage could assist in catching such problems in patches.

In this paper, we propose *protocol-level constraint-guided exploration*, a new approach to automatically generate vulnerability point reachability predicates with high coverage and in a compact form, for a given vulnerability point and an initial exploit message. Our approach has 3 main characteristics: 1) it is *constraint-guided* (i.e., instead of using black-box probing), 2) it works at the *protocol level*, and 3) it effectively *merges* explored execution paths to remove redundant exploration. Below we give an overview of the three characteristics:

Constraint-guided. Black-box probing techniques [14, 16] fundamentally suffer from the well-known limitations of blackbox fuzzing, where constraints that are satisfied by only a very small fraction of the input space are very hard to discover. For example, a conditional statement such as `if (FIELD==10) || (FIELD==20) then exploit, else safe`, creates two paths to the vulnerability point. If FIELD had value 10 in the original exploit message, the signature still needs to discover the alternate path when `FIELD == 20`, which would require 2^{30} random probes on average to discover. If the signature only covers the original path, where FIELD had value 10, then the attacker can easily evade detection by changing FIELD to be 20. Thus, we propose to use a constraint-guided approach by monitoring the program execution, performing symbolic execution to generate path predicates, and generating new inputs that will go down a different path by taking a different branch at a branching point. Such approach would only require two tests to discover the above condition.

At a first glance, our constraint-guided exploration may resemble recent work in symbolic execution for automatic test case generation [13, 19]. However, there are two significant differences. First, the problem addressed is different. In automatic test case generation, the goal is to cover all program paths or at least maximize block or branch coverage (i.e., exploring every basic block or both the *true* and *false* branches at each branching point), while in automatic signature generation the goal is to find all paths that lead to the vulnerability point. Moreover, in automatic test case generation, one is often only interested in finding *some* inputs which make the program execution reach a particular basic block or branch, whereas in automatic signature generation we are interested in finding *all* inputs which make the program execution reach the vulnerability point (to minimize false negatives). Second, those techniques explore paths one at a time. Given that the number of paths is often exponential in the number of conditions or even infinite, such approaches simply do not scale in our setting. In fact, in Bouncer [14] the authors acknowledge that they wanted to use a constraint-guided approach but failed to do so due to the large number of paths that need to be explored and thus had to fall back to the black-box probing approach.

To make the constraint-guided exploration feasible and effective and to make the resulting vulnerability point reachability predicate compact, we have incorporated two other key characteristics into our approach as described below.

Protocol level. Previous symbolic execution approaches generate what we call *stream-level conditions*, i.e., constraints that are evaluated directly on the stream of input bytes. Such stream-level conditions in turn generate *stream-level signatures*, which are also specified at the byte level. However, previous work has shown that signatures are better specified at the protocol level instead of the byte level [16, 34] as the vulnerability is often best specified at the protocol level. We call such signatures *protocol-level signatures*.

Our contribution here is to show that, to make the constraint-guided approach feasible it is fundamental to lift stream-level conditions to *protocol-level conditions*, so that they operate on protocol fields rather than on the input bytes, as using constraints at the protocol-level hugely reduces the number of paths to be explored compared to using stream-level conditions. Protocol-level conditions reduce the space that needs to be explored for two main reasons. First, the parsing logic often introduces huge complexity in terms of the number of execution paths that need to be analyzed. For example, our experiments show that with HTTP vulnerabilities 99% of all constraints are generated by the parsing logic. While such constraints generated by the parsing logic need to be present in the stream-level conditions, they can be removed in the protocol-level conditions. Second, the constraints introduced by the parsing logic fixes the field structure to be the same as in the original exploit message, for example fixing variable-length fields to have the same size as in the original exploit message, and fixing the field sequence to be the same as in the exploit message (when protocols such as HTTP allow fields to be reordered). Thus, new paths generated by changing the length of the variable-length fields or by reordering the fields, would also have to be explored if we use stream-level conditions. Otherwise, the resulting signature would be very easy to evade by an attacker by applying those small variations to the field structure of the original exploit message. For the same reason, expressing the vulnerability point reachability predicate at the protocol level naturally makes it much more compact than at the stream level as it automatically accounts for variable-length fields and field reordering. Finally, as a side benefit, expressing the vulnerability point reachability predicate at the protocol level also makes it much easier to understand by humans.

Merging execution paths. In Bouncer, the authors propose to use a vulnerability point reachability predicate that is a disjunction (i.e., an enumeration) of all the discovered paths that lead to the vulnerability point. Such vulnerability point reachability predicate has two main problems. First, it suffers from the combinatorial explosion problem since the number of paths is exponential on the number of constraints. Second, it generates signatures that quickly explode in size. ShieldGen tries to avoid this problem by removing fields whose value do not affect whether the vulnerability point is reached. Their problem is that to confidently decide whether the value of a field does not matter to reach the vulnerability point, they would need to probe all the values of the field, which does not scale with large fields (e.g., 32-bit). Thus, they revert to heuristics to sample the field values and thus can introduce false positives by removing fields for which a few values allow to reach the vulnerability point.

To overcome those limitations, we utilize the observation that the program execution may fork at one condition into different paths for one processing task, and then merge back to perform another task. For example, a task can be a validation check on the input data. Each independent validation check may generate one or multiple new paths (e.g., looking for a substring in the URI generates many paths), but if the check is passed then the program moves on to the next task, which usually merges the execution back into the original path. Thus, in our exploration, we use a *protocol-level exploration graph* to identify such potential merging points and give priority to exploring paths that are likely to merge into previously explored paths that reach the vulnerability point. This helps alleviate the combinatorial exploration problem, and allows our exploration to quickly reach high coverage.

Endeavour. We have designed and developed our approach into Endeavour, a system that automatically generates high coverage, yet compact, vulnerability point reachability predicates. Our evaluation demonstrates the effectiveness of our approach. For example,

compared to Bouncer, our vulnerability point reachability predicate for the Microsoft SQL Server vulnerability contains three conditions while the one obtained by Bouncer contains more than a hundred [14]. In addition, our vulnerability point reachability predicate for the GHttpd vulnerability is empty, meaning that any input that can be parsed reaches the vulnerability point, which we have verified using the source code. Bouncer generates a vulnerability point reachability predicate with more than 2000 conditions for the same vulnerability, and requires the exploit to be a GET request. Compared to ShieldGen, our vulnerability point reachability predicate for the Microsoft DCOM RPC vulnerability discovers conditions that are required to reach the vulnerability point and were missed by ShieldGen. For example, the number of `ContexItem` elements in the `Bind` request, represented by the `NumCtxItems` field should not exceed 20. Such condition was missed by ShieldGen [4]. Thus, their vulnerability point reachability predicate will flag safe inputs as exploits, introducing false positives.

2. Problem Definition and Approach Overview

In this section, we first introduce the problem of automatic generation of protocol-level vulnerability point reachability predicates, and then give the overview of our approach.

2.1 Problem Definition

Protocol-level vulnerability point reachability predicate. Given a parser implementing a given protocol specification, the vulnerability point, and an input that exploits the vulnerability at the vulnerability point in a program, the problem of automatic generation of protocol-level vulnerability point reachability predicates is to automatically generate a predicate function F, such that when given some input mapped into field structures by the parser, F evaluates over the field structures of the input: if it evaluates to *true*, then the input is considered to be able to reach the vulnerability point, otherwise it is not.

Note that in the problem domain of automatic generation of protocol-level signatures (and thus automatic generation of protocol-level vulnerability point reachability predicates), it is assumed that we are given a parser implementing a given protocol or file specification consistent with the one implemented in the vulnerable program. The parser given some input data can map it into the field structures, according to the protocol specification. This assumption can be addressed in many practical ways and we consider it outside the scope of this paper. For example, such parser is available for common protocols (e.g., Wireshark [6]), and many commercial network-based IDS or IPS have such a parser built-in. In addition, the parser may be manually constructed [16] using domain-specific languages for protocol specifications [9, 28], which allow building a generic parser that takes as input multiple protocol specifications. Also, recent work has proposed to automatically extract the protocol specification from the program binary [12, 25, 32]. Such work can be used when the protocol used by the vulnerable program has no public specification or its implementation deviates from the specification.

Note also that in our problem setting, we assume that an initial exploit message is given, as in the previous work of automatic signature generation. In addition, we assume that the vulnerability point is given as we consider identifying the vulnerability point given an exploit message is an orthogonal problem and out of scope for this paper. Such vulnerability point may be identified using previous techniques such as [14, 26].

Running example. Figure 1 shows our running example. We represent the example in C language for clarity, but our approach operates directly on program binaries. Our example represents a basic HTTP server and contains a buffer-overflow vulnerability. In the

```
01) void service() {
02)   char vulBuf[128], msgBuf[4096];
03)   char lineBuf[1024], method[256];
04)   char uri[256], ver[256];
05)   int nb=0, i=0, is_cgi=0, sockfd=0;
06)
07)   nb=recv(sockfd, msgBuf, 4096, 0);
08)   for(i = 0; i < nb; i++) {
09)     if ((msgBuf[i] == '\n') ||
10)       (msgBuf[i] == '\r'))
11)         break;
12)     else
13)       lineBuf[i] = msgBuf[i];
14)   }
15)   lineBuf[i] = '\0';
16)   sscanf(lineBuf,"%255s %255s %255s",
17)     method, uri, ver);
18)   if (strcmp(method, "GET") == 0 ||
19)       strcmp(method, "HEAD") == 0)
20)   {
21)     if strncmp(uri,"/cgi-bin/",9)==0
22)       is_cgi = 1;
23)     else
24)       is_cgi = 0;
25)     if (uri[0] != '/')
26)       return;
27)     strcpy(vulBuf, uri);
28)   }
29)   else if (strcmp(ver,"HTTP/0.9")==0
30)       && strcmp(ver,"HTTP/1.0")==0)
31)     printf("Invalid method");
32) }
```

Figure 1: Our running example.

example, the `service` function receives and processes the network requests. It uses a loop to copy the first input line into the variable `linebuf` before parsing it into several variables (lines 7-17). The first line in the exploit message includes the method, the URI of the requested resource, and the protocol version. If the method is GET or HEAD (lines 18-19), and the first character of the URI is a slash (line 25), then the vulnerability point is reached at line 27. The size of `vulBuf` is not checked by the `strcpy` function on line 27. Thus, a long URI can overflow the `vulBuf` buffer.

In this example, the vulnerability point is at line 27, and the vulnerability condition is that the local variable `vulBuf` will be overflowed if the size of URI field in the received message is greater than 128. Therefore, for this example, the vulnerability point reachability predicate is:

```
(strcmp(FIELD_METHOD,"GET") == 0 ||
strcmp(FIELD_METHOD,"HEAD") == 0)
URI[0] != '/'
```

while the vulnerability condition is: `length(FIELD_URI) > 128`, and the optimal protocol-level signature is the conjunction of the two. It is important to note that ShieldGen cannot produce such a vulnerability point reachability predicate because for each field, their blackbox probing approach only checks whether the field needs to have the same value as in the exploit message to exploit the vulnerability or not. For example, given a exploit message that was a `GET` request, it is nearly impossible for their black-box probing approach to discover that a `HEAD` request could exploit the program as well, thus introducing false negatives. In contrast, our results will show that Endeavour is able to accurately identify such conditions.

Note that even though this running example is on memory-safety type of vulnerability, the problem of identifying vulnerability point reachability predicates is general and applies to any type of vulnerability where one is interested in finding all the paths that may lead to the vulnerability point.

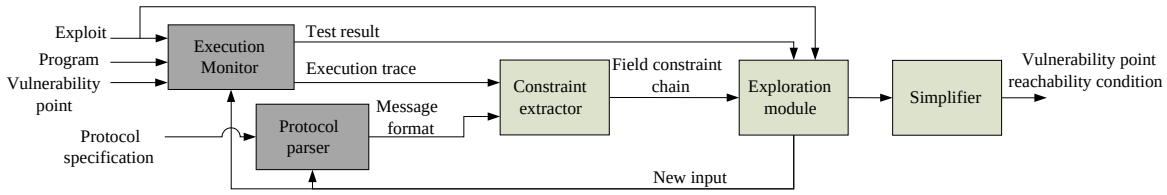

Figure 2: Endeavour architecture overview. The darker color modules are given, while the lighter color components have been designed and implemented in this work.

2.2 Approach and Architecture Overview

In this paper we propose a new approach to generate high coverage, yet compact, vulnerability point reachability predicates, called *protocol-level constraint-guided exploration*, which as our experiments demonstrate, achieves significantly better results than previous work. The architecture of our system, Endeavour, is shown in Figure 2. It comprises of two main components: the *constraint extractor* and the *exploration module*; an additional component, the *simplifier*; and two off-the-shelf assisting components: the *execution monitor* and *the parser*. The overall exploration process is an iterative process, controlled by the exploration module. The exploration module maintains the exploration state in a special graph, the *protocol-level exploration graph*, which we have designed specifically to enable effective exploration.

For each iteration of the exploration process, the exploration module uses the current protocol-level exploration graph to generate a new input that will lead the program execution to explore a new edge in the protocol-level exploration graph. The new input is then sent to the vulnerable program, which is run inside the execution monitor. The execution monitor oversees the program execution as it processes the input and outputs an execution trace, which summarizes the execution. It also outputs whether the input made the program execution reach the vulnerability point (e.g. the test result). Then the parser extracts the message format for the new input, according to the protocol specification.

Next, given the message format and the execution trace, the constraint extractor raises the stream level path predicate to the protocol level and removes unnecessary parsing conditions. The constraint extractor outputs the *field constraint chain* (Section 3). Finally, the exploration module adds the field constraint chain to the current protocol-level exploration graph performing a merging step to remove redundant paths. At that point, the exploration module generates a new input that will allow exploring another edge in the updated protocol-level exploration graph and the whole process repeats.

The process is started with the initial exploit message and runs iteratively until the termination condition is satisfied (Section 4). At that point the exploration module produces the intermediate vulnerability point reachability predicate, which is simplified by the simplifier into the final vulnerability point reachability predicate. The exploration module is detailed in Section 4.

3. Extracting the Protocol-level Path-predicate

In this section we present the constraint extractor, which given an execution trace, produces a protocol-level path-predicate.

Background: path predicate. Given the execution trace of the program processing an input, one can perform symbolic execution with the input represented as a symbolic variable and extract the *path predicate*, i.e., the predicate that an input needs to satisfy to take the program execution to follow the same path as in the execution trace. This path predicate is essentially the conjunction of all branch conditions dependent on the symbolic input in the given path. The concept of path predicate and how to compute it

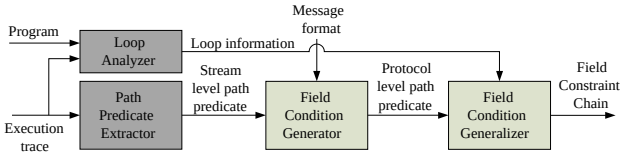

Figure 3: Constraint Extractor Architecture. The darker color modules are given, while the lighter color components have been designed and implemented in this work.

is well understood and has been widely used in previous work including vulnerability signature generation [10, 15] and automatic test case generation [13, 19]. Thus, we refer the interested reader to these previous work for details. Here we simply use an off-the-shelf assisting component *path predicate extractor* to generate the path predicate given an execution trace. Note that to deal with each branch condition in the path individually, the path predicate extractor computes a dynamic slice of each branch condition in the original path. Each slice contains all the statements that were used to generate the branching condition from the symbolic input. Then each slice is simplified to obtain a smaller but equivalent condition. The details of this process are presented in [7].

Key goals and overview. The path predicate used in previous work is at the stream level, i.e., the conditions are on raw bytes of the input, called *stream-level conditions* (where each byte of the input is represented as a symbolic variable indexed by its byte offset in the input, e.g., `INPUT[x]`, called a *byte symbol*). To enable constraint-guided exploration, we need to lift the path predicate to the protocol level, i.e., the conditions are instead on field variables of the input, called *protocol-level conditions* (where each byte of the input is represented as a symbolic variable indexed by its field name and its byte offset in the field, e.g., `FIELD_FieldName[x]`, called a *field symbol*). To make the distinction clear, we refer to the path predicate at the stream level the *stream-level path-predicate*, and the path predicate at the protocol level the *protocol-level path-predicate*. To accomplish this, the key contributions of the constraint extractor are, given the stream-level path-predicate, how to lift it to the protocol level (achieved by the *field condition generator*), and how to further generalize it by removing the parsing conditions (achieved by the *field condition generalizer*). Finally, to assist the construction of the protocol-level exploration graph (explained in Section 4), the constraint extractor outputs the *field constraint chain*, which is the protocol-level path-predicate annotated with the EIP for each branch condition and formed in an ordered chain of branch conditions using the same order as in the execution path.

3.1 The Field Condition Generator

Given the stream-level path-predicate generated by the path predicate extractor and the message format of the input given by the parser, the field condition generator outputs a protocol-level path-predicate. It performs this in two steps. First, it translates each byte

symbol `INPUT[x]` in the stream-level path-predicate into a field symbol `FIELD_fieldname[x - start(fieldname)]` using the mapping produced by the parser. Second, it tries to combine symbols on consecutive bytes of the same field. For example, the stream-level path-predicate might include the following condition: `(INPUT[6] << 8 | INPUT[7]) == 0`. If the message format states that inputs 6 and 7 belong to the same 16-bit `ID` field, then the condition first gets translated to `(FIELD_ID[0] << 8 | FIELD_ID[1]) == 0` and then it is converted to `FIELD_ID == 0` where `FIELD_ID` is a 16-bit field symbol.

Benefits. This step lifts the stream-level path-predicate, at the byte level, to the protocol-level path-predicate, at the protocol level. By doing so, we break the artificial constraints that the stream-level path-predicate imposes on the position of fields inside the exploit message. For example, protocols such as HTTP allow some fields in a message (i.e., all except the Request-Line/Status-Line) to be ordered differently without changing the meaning of the message. Thus, two equivalent exploit messages could have the same fields ordered differently and a byte-level vulnerability point reachability predicate generated from one of them would not flag that the other also reaches the vulnerability point. In addition, if variable-length fields are present in the exploit message, changing the size of such fields changes the position of all fields that come behind it in the exploit message. Again, such trivial variation of the exploit message could defeat stream-level conditions. Thus, by expressing constraints using field symbols, protocol-level conditions naturally allow a field to move its position in the input.

3.2 The Field Condition Generalizer

The field condition generalizer takes as input the protocol-level path-predicate generated by the field condition generator and the loops extracted by the *loop analyzer*, an off-the-shelf assisting component, and outputs a field constraint chain where the parsing-related conditions have been removed.

The *loop analyzer* extracts the loops in the execution trace using the static information extracted from the vulnerable program. The loop analyzer implements standard loop detection analysis algorithms [8] and outputs for each loop: the loop head, the loop body, and the loop exit conditions.

Motivation and Intuition. As shown in Section 3.1, by lifting the stream-level path-predicate to the protocol-level path-predicate, we allow the vulnerability point reachability predicate to cover additional variants where each field may be at a different position, e.g., when a variable-length field changes size. However, the parsing conditions that remain in the protocol-level path-predicate still over-constrain the variable-length fields, forcing them to have some specific size (usually the same as in the exploit message). Thus, to allow the vulnerability point reachability predicate to handle exploit messages where the variable-length fields have a size different than in the original exploit message, we need to remove such parsing conditions. In addition, for some protocols such as HTTP, the number of parsing conditions in a single protocol-level path-predicate can range from several hundreds to a few thousands. Such a huge number of unnecessary conditions would blow up the size of the vulnerability point reachability predicate and negatively impact the exploration that we will present in Section 4.

In this paper, we term parsing conditions to be conditions used by the program to identify the end of variable-length fields in a received message. These are the conditions that over-constrain variable-length fields to have a specific size. Other conditions generated by the program to check the value of a field (e.g., that a protocol version is supported, or that a maximum field value is not exceeded) do not over-constrain the variable-length fields and thus need to be kept in the protocol-level path-predicate.

Previous work on protocol reverse-engineering [12, 25, 32] has identified three main elements that protocols use to find the end of variable-length fields: separators (also called delimiters), length fields, and array counter fields. Their results show that those protocol elements are often used in programs to control the number of times that parsing loops iterate. Such usage is the reason why parsing conditions constrain the size of variable-length fields to be the same one as in the exploit message. Such behavior can be observed in our running example, where the loop at lines 14 to 20 is trying to locate the end of the first line of the message, which is variable-length and marked by the separator Carriage Return + Line Feed (`"\r\n"`). The `if` conditionals at lines 15-16 generate the following conditions in the protocol-level path-predicate: `(FIELD_METHOD[0]` $\neq$ `'\n')` $\wedge$ `(FIELD_METHOD[0]` $\neq$ `'\r')` $\wedge$ `,...,`$\wedge$`(FIELD_VERSION[x] == '\n')`

A new input with an extra byte in the URI field would have the Carrier Return character at position `x+1` rather than at position `x` and thus would not match the previous predicate since for this new input `FIELD_VERSION[x]` $\neq$ `'\n'`. Such input would be considered not to reach the vulnerability point, thus creating a false negative.

```
if (count > 64) then abort();
for (i = 0; i < count; i++) {
  /* Code to parse a record here */
}
```

Figure 4: Example of array parsing loop

The same occurs with length and array counter fields. For example, in the code snippet in Figure 4 where `count` is a field directly copied from the exploit message, representing the number of records in a variable-length array, if `count` had value 3 during the execution, the resulting conditions would be: `(count` $>$ `0)` $\wedge$ `(count` $>$ `1)` $\wedge$ `(count` $>$ `2)` $\wedge$ `(count` $\leq$ `3)`. Clearly, such loop constrains the `count` variable to have value 3. All other length and array counter fields that are used to control the number of iterations in a loop, become similarly constrained. Thus, a new exploit with four elements in the array, instead of 3, would not satisfy the condition `count` $\leq$ `3` and would be considered safe.

Such exploit variants that modify the size of variable-length fields are already in use. For example, tools such as the Metasploit framework [5] can create such exploits. To avoid such exploit variants to evade detection, the parsing conditions in the protocol-level path-predicate need to be identified and removed. We use two different techniques to remove the parsing conditions: function summaries for well-defined functions, and examining the loop exit conditions for other functions.

We use function summaries for well-known functions, most often for functions that operate on strings. Such function summaries have been proposed to limit the combinatorial explosion of paths in a program [11, 18]. In addition, function summaries expose conditional dependencies that are otherwise hidden by function calls. We describe our function summaries implementation in Section 5. Next, we present our loop analysis technique to remove parsing conditions, which can be used regardless of whether the program uses well-known functions for parsing the protocol messages.

Removing the parsing conditions. The field condition generalizer takes as input the information about the loops in the execution trace, extracted by the loop analyzer. For each loop it includes the loop head, the loop body, and the loop exit conditions. In addition, the execution trace contains information about which loop exit conditions use input data.

For each loop which contains exit conditions that use input data, the field condition generalizer removes those exit conditions if they

compare against the value of a separator (as provided in the protocol specification) or if the exit condition generates conditions that differ by a constant increment/decrement in each iteration. Such conditions map to the parsing conditions used by the program to identify the end of the variable-length fields.

In our running example, the loop at lines 14 to 20 is used by the program to identify the end-of-message separator, and contains an exit condition that depends on input data and on the value of the separator at line 15. From the HTTP protocol specification we would obtain that the Carriage-Return ($\backslash n$), the Line-Feed ($\backslash r$), the space and the tab characters can be used as separators. Thus, the conditions generated from the `if` conditional at line 15 would be identified as parsing conditions because they compare input data with a separator value. Consequently, they would be removed from the protocol-level path-predicate.

In the loop snippet in Figure 4, all the conditions generated by the loop exit condition would be identified as parsing conditions because the generated conditions: `(count` $>$ `0)` $\wedge$ `(count` $>$ `1)` $\wedge$ `(count` $>$ `2)` $\wedge$ `(count` $\leq$ `3)` differ by a constant increment . However, the condition generated by the first `if` conditional does not belong to a loop exit condition and thus the field condition generalizer would not remove it. Intuitively, this would keep among others, conditions that programs might perform to validate the value of the length and array counter fields before using them in a loop, which are not part of the parsing process.

Symbolic Memory Addresses The field condition generalizer also performs some special handling if there is any field condition that depends on a memory read from a symbolic address, which are often the result of a translation table. Such translation tables usually occur with string operations that do upper/lower-case or ASCII/Unicode transformations. For example, one such condition could be: `mem[FIELD_URI[6] + 0x81d9440] == 0x49`, where the input data is used to look up into a mapping table whose base address is 0x81d9440. Clearly, with a different value of the field URI, the address being accessed would be different and we would not know how to handle it. We address this problem in two different ways depending on the scenario. The first case is if the symbolic memory address belongs to a static table in the data segment. In this case, we simply incorporate the table into the signature. In the second case, the symbolic memory address does not belong to a static table. In this case, we take the conservative approach and fix the symbol to have the same value as in the exploit message. For example if FIELD_URI[6] had value 0x69 in the exploit message, then the above condition would be transformed into: FIELD_URI[6] == 0x69. Such condition might over-constrain the value of the symbol and we address this case in two ways. First, most times such translation tables appear in well-defined functions such as `sscanf`. Thus, those conditions are removed by the function summaries. In addition, during exploration, new values for the over-constrained symbol will be explored and if the vulnerability point is still reached for those values, then the conditions on the symbol will be relaxed and sometimes even eliminated.

The field constraint chain. To assist the construction of the protocol-level exploration graph (explained in Section 4), the constraint extractor constructs the *field constraint chain* using the generated protocol-level path-predicate. A field constraint chain is an enhanced version of the protocol-level path-predicate where each branch condition is annotated with the EIP of the corresponding branching point, and these annotated branch conditions are put in an ordered chain using the same order as they appear in the execution path.

4. Execution-guided Exploration

In this section we present the exploration module. Given the field constraint chain produced by the constraint extractor from the execution trace obtained using the given exploit message, the exploration module performs protocol-level constraint-guided exploration to generalize the constraints and output the vulnerability point reachability predicate (VPRP).

Two unique characteristics enable our approach to extract high coverage VPRPs. First, our exploration works on a special graph, the *protocol-level exploration graph*; the nature of this graph enables us to drastically cut down on the exploration space by grouping many execution paths into equivalent paths, so that they do not need to be explored separately. Second, as we discover new paths and add them to the protocol-level exploration graph, we merge the new paths with previously discovered paths as much as possible, thus further reducing the exploration space. In this section, we first introduce the protocol-level exploration graph and explain its benefits and our intuition of merging paths, and then we describe the overall exploration process and how to extract the final vulnerability point reachability predicate. Note that as we explained in Section 1, our exploration problem is markedly different from previous work on automatic test case generation (such as [13,19]) and thus their techniques are inadequate to address our task here.

4.1 The Protocol-level Exploration Graph

The protocol-level exploration graph. The explorer dynamically builds a *protocol-level exploration graph* as the exploration progresses. In a protocol-level exploration graph, each node represents an input-dependant branching point (i.e., a conditional jump) in the execution, which comprises the protocol-level condition and the address of the program's instruction (*eip*) for this branching point. Each node can have two edges representing the branch taken if the node's condition evaluated to true (`T`) or false (`F`). We call the node where the edge originates the *source node* and the node where the edge terminates the *destination node*. If a node has an *open edge* (i.e, one edge is missing), it means that the corresponding branch has not yet been explored.

The reason why nodes are labeled with both the condition and the eip (as shown in Figure 5A) is that the same branching point in an execution trace can generate different conditions depending on the content of the operands. For example, a compare `cmp %eax,%ecx` instruction followed by a jump-if-above instruction `ja 0x7e12abc` can generate both conditions: `FIELD_0 > 5` and `FIELD_1 > 7` at different points of a program by loading the content of the `eax` and `ecx` registers with different values.

Differences with automatic test case generation. Using a protocol-level exploration graph is significantly different from the traditional constraint-based exploration used in automatic test case generation approaches such as DART [19] and EXE [13]. First, in a protocol-level exploration graph the condition associated with each node is at the protocol level, whereas in the traditional constraint-based exploration, the conditions are at the stream level. Second, in a protocol-level exploration graph, the branching points corresponding to parsing conditions have already been removed, while they are present in the traditional constraint-based exploration.

Benefits of the protocol-level exploration graph. The above differences showcase the two fundamental benefits of using the protocol-level exploration graph: 1) the exploration space is significantly reduced, and 2) it becomes easy to merge paths, which in turn allows to further reduce the exploration space, and to generate compact vulnerability point reachability predicates.

The reduction in the exploration space is achieved in two ways. First, the protocol-level exploration graph does not contain the large number of parsing conditions that are usually present in the individual stream-level path-predicates because they have already been removed by the constraint extractor. Thus, we avoid overly constraining the input fields, which makes the constraints more ro-

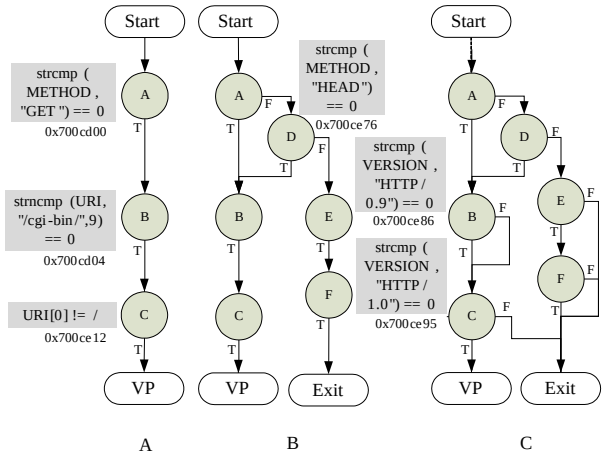

Figure 5: Building the protocol-level exploration graph for our running example.

bust to evasion attacks (e.g., reordering fields). Second, because the node's condition is at the protocol level then each path in the protocol-level exploration graph represents many paths in the traditional constraint-based exploration approach. For example, all paths generated by changing the size of the variable-length fields or by reordering fields in a message, are summarized in the protocol-level exploration graph, and thus do not need to be explored.

The second benefit is that having the node's condition at the protocol level allows to effectively merge paths (as explained below), even when the paths have been generated by multiple exploit messages that may contain the same variable-length fields with different sizes, or the same fields but in different order. For example, when handling an HTTP request, a server could first parse the message and next check if the `Host` field contains a private address. This check could be always done right after the parsing regardless of the position of the `Host` field in the message, which can change. Thus, the stream-level conditions generated for one message would not match a different message where the `Host` field is at a different position, but the protocol-level conditions would, thus allowing to merge the paths.

Intuition for merging execution paths. Another important characteristic of our approach is that as we discover new paths and add them to the protocol-level exploration graph, we merge the new paths with previously discovered paths in the protocol-level exploration graph as much as possible, and thus further reduce the exploration space. The key intuition behind merging paths is that it is common for new paths generated by taking a different branch at one node, to quickly merge back into the original path. This happens because programs may fork execution at one condition for one processing task, and then merge back to perform another task. One task could be a validation check on the input data. Each independent check may generate one or multiple new paths (e.g., looking for a substring in the URI generates many paths), but if the check is passed then the program moves on to the next task (e.g., another validation check), which usually merges the execution back into the original path. For example, when parsing a message the program needs to determine if the message is valid or not. Thus, it will perform a series of independent validity checks to verify the values of the different fields in the message. As long as checks are passed, the program still considers the message to be valid and the execution will merge back into the original path. But, if a check fails then the program will move into a very different path, for example sending an error message.

Next, we present the exploration process. We illustrate the different steps using Figure 5 which represents different steps in the exploration process using our running example. Note that, the A–F node labels are not really part of the protocol-level exploration graph but we add them here to make it easier to refer to the nodes.

4.2 The Exploration Process

Overall process. Figure 6 shows the architecture of the exploration module. The *explorer* is the core of the exploration; it maintains the protocol-level exploration graph and directs the other components. At the beginning, the explorer initializes the protocol-level exploration graph with the field constraint chain output by the constraint extractor from the execution trace generated using the initial exploit message. This is shown in Figure 5A where the original path starts at the `Start` node and contains the three nodes introduced by lines 18, 21, and 25 in our running example. The sink of the original field constraint chain is the vulnerability point node (`VP`).

The remaining exploration is an iterative process that comprises of six steps: (1) Given the current protocol-level exploration graph, the prioritization engine decides which node with an open edge to explore next; (2) For the selected node's open edge, the *input generator* generates a new input that will lead the program execution to reach that node and follow the selected open edge; (3) The new input is passed to the *replay tool* that sends it to the execution monitor; (4) The execution monitor checks whether the program execution has reached the vulnerability point and outputs an execution trace[1]; (5) The constraint extractor extracts the field constraint chain from the execution trace; (6) The explorer inserts the new field constraint chain into the current protocol-level exploration graph and obtains an updated protocol-level exploration graph. Then the prioritization engine selects a new open edge from the updated protocol-level exploration graph and the whole process repeats. The process repeats until all open edges have been explored or a user-specified timeout expires. When the exploration stops, the explorer extracts the VPRP and passes it to the simplifier who performs different simplifications and outputs the final VPRP.

In the remaining of this section, we first provide more details for steps 2 and 6, and then we explain how to prioritize which node to explore next (step 1). Step 5 has already been explained in Section 3 and steps 3 and 4 do not require further explanation. Finally we explain how to extract the final VPRP.

Adding the new path to the exploration graph. To insert a new field constraint chain into the protocol-level exploration graph, the explorer starts merging from the bottom until it finds a node that it cannot merge, either because it is not in the protocol-level exploration graph already, or because the predecessor in the new path is not the same one as in the protocol-level exploration graph. To check if the node is already in the protocol-level exploration graph, the explorer checks if the node to be inserted is equivalent (same EIP and equivalent condition) to any other node already in the protocol-level exploration graph. We call the last node that can be merged from the bottom the *join node*. At that point, the explorer starts merging from the top until it finds a node that it cannot merge, either because it is not in the protocol-level exploration graph or because the successor is not the same one in the new path as in the protocol-level exploration graph. We call the last node that can be merged from the top the *split node*. Next, we add the sequence of nodes in the new path between the join node and the split node as a new sequence of nodes in the protocol-level exploration graph.

For example, in Figure 5B, the prioritization engine selects the false branch of node A to be explored, which discovers the fol-

[1]The execution monitor decides that the vulnerability point has not been reached if an answer is sent or a timeout expires before seeing the address of the vulnerability point.

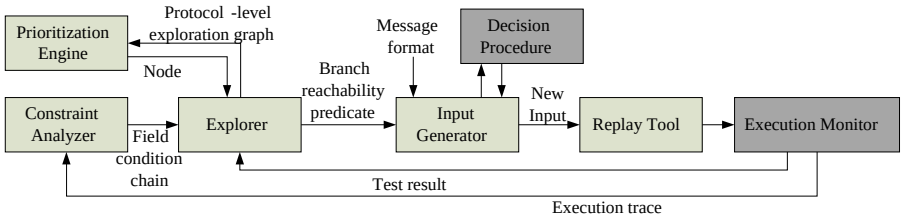

Figure 6: Exploration module architecture. The darker color modules are given, while the lighter color components have been designed and implemented in this work.

lowing new path to the VP: $\overline{A} \wedge D \wedge B \wedge C$. The explorer starts merging the new path from the bottom, first merging node C, then merging node B. At that point it cannot merge node D because there is no equivalent node in the graph. Thus, node D is the join node. Then, it starts merging from the top. First it merges node A in the protocol-level exploration graph with node $\overline{A}$ in the new path using the false branch of node A. Then, it cannot merge node D because there is no equivalent node in the protocol-level exploration graph. Thus, node D is also the split node. Since there are no more nodes in the new path between the split node and the join node (both are node D), then the explorer is done adding the path.

Generating a new input for a new branch. We define a *node reachability predicate* to be the predicate that summarizes how to reach a specific node in the protocol-level exploration graph from the `Start` node. Thus, the node reachability predicate summarizes all the paths that allow to reach the node in the current protocol-level exploration graph. Similarly, we define a *branch reachability predicate* to be the predicate that summarizes how to traverse a specific branch (true of false) from a specific node. Intuitively a branch reachability predicate is the conjunction of a node reachability predicate with the node's condition (if we want to traverse the true branch), or the negation of the node's condition (to traverse the false branch). The branch reachability predicate captures how the execution reaches the node and then takes the desired branch in the current protocol-level exploration graph.

To compute a new input that traverses the specific open edge selected by the prioritization engine, the explorer first computes the branch reachability predicate. Then, the input generator creates a new input that satisfies the branch reachability predicate.

To compute the branch reachability predicate, the explorer first computes the node reachability predicate. The node reachability predicate is essentially the weakest-precondition (WP) [17] of the source node of the open edge over the protocol-level exploration graph—by definition, the WP captures all paths in the protocol-level exploration graph that allow to reach the node. Then, the explorer computes the conjunction of the WP with the node's condition or with the negated condition depending on the selected branch. Such conjunction is the branch reachability predicate, which is passed to the input generator.

For example, in Figure 5B if the prioritization engine selects the false branch of node D to be explored next, then the branch reachability predicate produced by the explorer would be: $\overline{A} \wedge \overline{D}$. Similarly, in Figure 5C if the prioritization engine selects the false branch of node B to be explored next, then the branch reachability predicate produced by the explorer would be: $(A \vee (\overline{A} \wedge D)) \wedge \overline{B}$.

The input generator generates a new input that satisfies the branch reachability predicate using a 3-step process. First, it uses a decision procedure to generate field values that satisfy the branch reachability predicate. If the decision procedure returns that no field values allow to reach that branch, then the branch is connected to the special `Unreachable` node. Second, it extracts the values for the remaining fields (not constrained by the decision procedure) from the original exploit message. Third, it checks the message format provided by the parser to identify any fields that need to be updated given the dependencies on the modified values (such as length or checksum fields). Using all the collected field values it generates a new input and passes it to the replay tool.

Prioritizing paths. To prioritize which path to explore next, the prioritization engine uses a simple weight scheme, where there are only two weights 0 and 1. The explorer assigns a weight of 1 to each newly added node. If it finds a node such that none of its edges can lead to the `VP` node in the current protocol-level exploration graph, then it changes the weight for that node and all its successors to 0. (e.g., node E and its successor F in Figure 5C). The intuition, is that if a new path does not quickly lead back to the `VP` node, then the message probably failed the current check or went on to a different task and thus it is unlikely to reach `VP` later. The explorer simply selects the next node with weight 1 where only one edge has been explored, until it exhausts that list.

After no more nodes with weight 1 have open edges, then the explorer recalculates the branch reachability predicate for all branches connected to the *Unreachable* node to check if they can be traversed now. This could happen if a new path had been discovered (after that branch was explored) that leads into the node that is the source of that branch. If any of those branches can be traversed now, then the exploration keeps iterating until no nodes with weight 1 are left. After finishing this step, the whole process is repeated with nodes with weight 0.

For example, in Figure 5C, after reversing the false branch of node E, the explorer discovers that none of the branches of node E lead to the VP node. At that point node E and its successor F are assigned weight 0, while the remaining nodes have weight 1. Thus, the false branch of node F is not explored until all other weight 1 nodes have no remaining open edges, meaning that it is the last branch explored in Figure 5C.

Extracting the vulnerability point reachability predicate. Once the exploration ends, the protocol-level exploration graph contains all the discovered paths leading to the vulnerability point. To extract the vulnerability point reachability predicate from the protocol-level exploration graph the explorer computes the node reachability predicate for the VP node, which as explained above corresponds to the WP of the VP node.

For our running example, represented in Figure 5D the extracted vulnerability point reachability predicate is: $(A \vee (\overline{A} \wedge D)) \wedge C$. Note that, the disjunction approach used by Bouncer would generate the following vulnerability point reachability predicate: $(A \wedge B \wedge C) \vee (\overline{A} \wedge D \wedge B \wedge C) \vee (A \wedge \overline{B} \wedge C)$. Such signature contains 10 conditions as opposed to 4 in our signature. Clearly, such signature is inefficient and can quickly explode in size.

Simplifier. After the explorer outputs the initial VPRP, the simplifier performs a simplification step to remove conditions that may be redundant. The simplifier checks whether a condition is already implied by any of its dominator conditions. For example, if a condi-

tion `FIELD > 5` has a single predecessor condition `FIELD > 8` then the condition `FIELD > 5` is redundant because it is implied by the predecessor. After removing such redundant conditions the simplifier outputs the final VPRP.

5. Implementation

In this section we provide some notable implementation details of our system. We have implemented the constraint extractor and the exploration module with 5000 lines of Ocaml code. Next, we present the remaining assisting components.

Execution Monitor. The execution monitor is an assisting component based on the QEMU emulator [2]. The execution monitor marks the inputs to the program as symbolic and keeps track of which parts of the program state become symbolic during execution. It outputs an execution trace that contains the executed instructions that operate on symbolic data, the content of the instruction's operands when the instruction was executed, and the associated symbolic information for each operand.

Function summaries. We have implemented function summaries in the execution monitor. In particular, we define a list of functions that we are interested in (e.g., string operations). When any of those functions is called, the execution monitor calls some special handling code which retrieves the function's arguments from the stack and checks whether they are symbolic. If any argument is symbolic, then all the arguments are stored, all outputs of the call are marked with new symbols, and any instructions that operate on symbolic data inside the function are ignored. Such processing allows us to obtain conditions that depend on the output of the call. For example our vulnerability point reachability predicate might contain conditions such as: strstr(FIELD_URI,"/../") == 0. In addition, for each function summary we have added a small piece of code that allows to reverse the function summary condition. Currently we have support for 4 groups of string functions: comparison functions such as strcmp, strncmp, strcasecmp, strncasecmp; substring functions such as strstr, strrstr; length functions such as strlen; and tokenization functions such as strtok.

Parser. Currently, we use Wireshark as our parser due to its support of a large number of protocols. We have defined our own tree-like field structure to which we map the field information provided in the Wireshark API. We are also studying the use of more general languages that provide a cleaner interface to the parser [9, 28].

6. Evaluation

In this section, we present the results of our evaluation. We first present the experiment setup, then the constraint extractor results and finally the exploration results.

Experiment setup. We evaluate our approach using 5 vulnerable programs, summarized in Table 1. The table shows the program, the protocol used by the vulnerable program, the type of protocol (i.e., binary or text), and the guest operating system used to run the vulnerable program. The exploits are either from the Metasploit Framework [5] or from SecurityFocus.com [3]. The first four vulnerabilities are buffer overflows affecting vulnerable HTTP servers (GHttpd, AtpHttpd) running on Linux, and services shipped by default in Microsoft Windows (DCOM RPC, SQL server). The last vulnerability affects the gdi32.dll library included in Windows XP. It is an integer overflow in the code to parse a Windows Media file (WMF), which causes an infinite loop.

Program	Protocol	Type	Guest OS
Win. DCOM RPC	RPC	Binary	Win. XP
Micros. SQL Server	Proprietary	Binary	Win. 2000
GHttpd	HTTP	Text	Red Hat 7.3
AtpHttpd	HTTP	Text	Red Hat 7.3
gdi32.dll	WMF file	Binary	Win. XP

Table 1: Vulnerable programs used in the evaluation.

6.1 Constraint Extractor results

In this section we evaluate the effectiveness of each step in the constraint extractor, which we measure by the number of unnecessary conditions that are removed at each step. For simplicity, we only show the results for the protocol-level path-predicate obtained by the constraint extractor from the execution trace generated by the original exploit. Note that, during exploration this process is repeated once per newly generated input. The four leftmost columns in Table 2 summarize the results. The *Original* column represents the number of input-dependant conditions in the original path constraint and is used as the base for comparison. The *Summaries* column shows the number of remaining conditions after conditions generated by the functions that have function summaries have been removed. The *Parsing* column shows the number of remaining conditions after the field condition generalizer removes the parsing conditions.

The results show the effectiveness of each step. First, the function summaries are very useful with the HTTP vulnerabilities, removing between 60% and 83% of the original conditions. This is because both servers use many standard string functions provided by the C library such as `strstr` or `strlen`. Surprisingly, thousands of conditions in the SQL Server are also removed by function summaries. This happens because the exploit message contains a long database name string, and the server uses the `strlen` function to find the end of the string, which results in a comparison to the null terminator for each character in the string. However, we have no function summaries for the private functions in the dynamically link libraries (DLLs) used by Windows to parse the RPC messages, thus no conditions are removed by the function summaries. Such equivalent conditions are mainly due to functions that are repeatedly called by the program and to redundant checks that the program performs.

Second, the results show that parsing conditions constitute 99% of all conditions for both HTTP vulnerabilities. These are due to loop scanning for the end of line delimiters. The number of parsing conditions removed in the DCOM RPC example is not large, but it is important to remove such conditions because otherwise they constrain the Context Item Array in the first message to the original 13 elements that it contained in the exploit message. The GDI original protocol-level path-predicate contains only four conditions, all on fixed-length fields, thus no parsing conditions are removed.

6.2 Exploration Results

The two rightmost columns in Table 2 show the results for the exploration phase. The *Exploration* column shows the number of nodes that can reach the vulnerability point, remaining in the protocol-level exploration graph after the exploration has finished. Finally, the *VPRP* column shows the final number of conditions in the VPRP after the explorer has extracted the VPRP and the simplifier has removed all redundant conditions from it.

Note that, previous work has just provided the number of conditions as a measure of their approach without providing detailed explanation about what those remaining conditions represent due to the large number of remaining conditions. Given the success of the constraint extractor at removing conditions in four of the five original protocol-level path-predicates and also to the small size of our conditions, we are the first to be able to show what the remaining conditions in the protocol-level path-predicate represent. We do that by labeling the nodes in the protocol-level exploration graph with the full protocol-level conditions.

	Constraint Extractor (Original exploit)			Exploration phase	
Vulnerable Program	**Original**	**Summaries**	**Parsing**	**Exploration**	**VPRP**
Windows DCOM RPC	732	732	662	659	423
SQL Server	2447	7	6	6	1
GHttpd	2498	420	5	3	3
AtpHttpd	6034	2430	11	19	13
GDI	4	4	4	5	5

Table 2: Summary of evaluation results. On the left the results for each step of the constraint extractor on the original exploit messages, and on the right the exploration results.

Performance summary. As a measure of the scalability of the approach we note that the average time to complete an iteration of the exploration process varies between 20 seconds (in the case of the GDI and SQL Server vulnerabilities) and 21 minutes (in the case of the Atphttpd vulnerability). Due to space limit, we do not provide the full detailed time breakdowns. Also, the performance can be significantly improved by a more optimized implementation.

MS SQL server. The parser shows that there are only two fields in the exploit message: the CMD (command) field and the DB (database name) field. The original protocol-level path-predicate returned by the constraint extractor contains 6 conditions: 4 on the CMD field and the other 2 on the DB field. Thus, the initial protocol-level exploration graph contains 6 nodes.

The exploration explores the open edges of those 6 nodes and finds that none of the newly generated inputs reaches the vulnerability point and no new nodes are discovered. Thus, the VPRP returned by the explorer contains the following 6 conditions:

```
(FIELD_CMD <= 9) ^ (FIELD_CMD != 9) ^
(2 <= FIELD_CMD <= 8) ^ (FIELD_CMD == 4) ^
(strcmp(FIELD_DB, "") != 0) ^
(strcasecmp(FIELD_DB, "MSSQLServer") != 0)
```

After the simplifier removes the redundant conditions, the resulting VPRP is

```
(FIELD_CMD == 4) ^
(strcmp(FIELD_DB, "") != 0) ^
(strcasecmp(FIELD_DB, "MSSQLServer") != 0)
```

Note that, the vulnerability condition for this vulnerability states that the length of the DB field needs to be larger than 64 bytes. Thus, the last two conditions in the VPRP are redundant and the final protocol-level signature would be:

```
(FIELD_CMD == 4) ^
length(FIELD_DB) > 64
```

GHttpd. The parser shows that there are three fields in the exploit message: the METHOD, the URI, and the VERSION. The original protocol-level path-predicate contains 5 conditions: 3 on the METHOD, 1 on the URI and 1 on the whole message. Thus, the initial protocol-level exploration graph contains 5 nodes. The exploration discovers five new paths to the vulnerability point and no new nodes. It turns out that exploring any branch of the original 5 nodes always leads to the vulnerability point. Thus, the VPRP returned by the simplifier is empty, meaning that any input that can be parsed would reach the vulnerability point. Figure 11 shows the optimal signature for the GHttpd vulnerability that we manually extracted from the source code. It contains a disjunction of 3 cases but a close look reveals that indeed any input allows to reach the vulnerability point. But, not every input exploits the program. The vulnerability condition varies depending on the path taken to the vulnerability but requires the METHOD field to be at least 166 bytes or the URI field to be at least 160 bytes (since the minimum IP address string has 6 bytes). Thus, in this case we extract the optimal VPRP.

AtpHttpd. As in the GHttpd exploit, there are three fields in the exploit message: the METHOD, the URI, and the VERSION. The original protocol-level path-predicate contains 11 conditions: 2 on the METHOD field and the other 9 on the URI field. Thus, the initial protocol-level exploration graph contains 11 nodes.

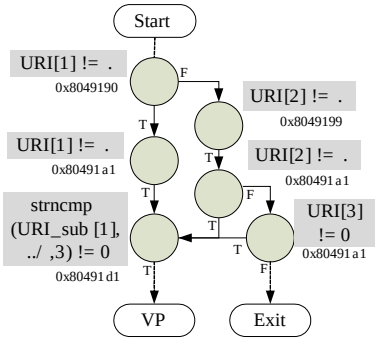

Figure 7: Part of the AtpHttpd protocol-level exploration graph corresponding to a strcmp(URI_sub[1], "..") function call inlined by the compiler.

The exploration discovers 8 new nodes in the protocol-level exploration graph The 8 newly discovered nodes correspond to different paths generated by the compiler inlining the following two function calls: `strcmp(URI_sub[1], "..")`, and `strcmp (URI_sub[2], "/..")`, where `URI_sub[1]`, and `URI_sub[2]` are the URI substrings starting at the second byte and third bytes respectively.

Figure 7 shows the relevant part of the protocol-level exploration graph for the first inlined call, which introduces 3 new nodes. The three nodes vertically aligned on the left were introduced by the original protocol-level path-predicate, and the others were discovered by the exploration. Note that, there are two pairs of equivalent conditions: `URI[1] != '.'` and `URI[2] != '.'` appear twice. Those equivalent conditions in turn generate two infeasible paths, that is, we cannot generate inputs that take the false branch for the latter conditions. We have found that such equivalent (and redundant) conditions often appear when the compiler inlines function calls.

The initial VPRP extracted from the protocol-level exploration graph contains 6 redundant nodes that are removed by the simplifier. The final VPRP is:

```
(strcasecmp(FIELD_METHOD,"get") == 0) ^
(FIELD_URI[0] == '/') ^
(FIELD_URI[1] != '/') ^
(strcmp(FIELD_URI_sub[1],"..") != 0) ^
(strncmp(FIELD_URI_sub[1],"../",3) != 0) ^
(strstr(FIELD_URI_sub[1],"/../") == 0) ^
(strcmp(FIELD_URI_sub[2],"/..") != 0) ^
(FIELD_URI[1] != 0) ^
```

where we have substituted the conditions inlined by the compiler with the equivalent function calls (lines 4 and 7 above) for readability. Figure 10 shows the optimal signature for this vulnerability, which we have manually extracted from the source code. Note that, our VPRP is almost optimal. Ignoring the last line in the op-

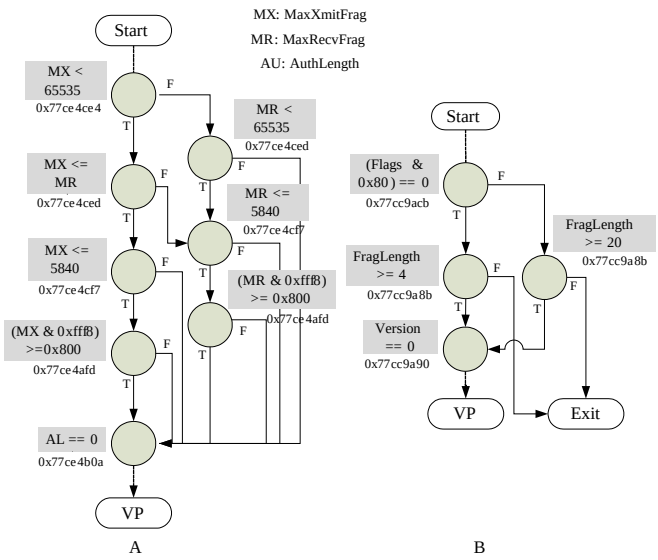

Figure 8: Parts of the DCOM RPC protocol-level exploration graph showing the newly discovered paths.

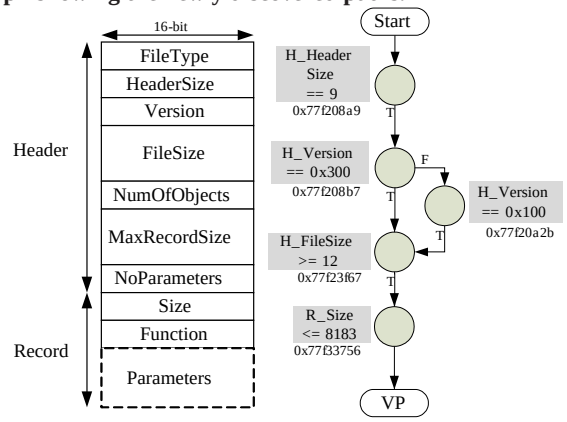

Figure 9: On the left, the format of the GDI exploit file. On the right the final GDI vulnerability point reachability predicate. There are 4 fields in the WMF file header that do not affect whether the VP is reached.

timal signature that represents the vulnerability condition, Endeavour only misses the disjunction starting with `isnotdir` because we currently don't model system calls such as `stat`, used to check whether a file exists in the server.

GDI. The original protocol-level path-predicate for the GDI exploit file contained 4 conditions. Figure 9 shows on the left the field structure for the exploit file, where an invalid value in the `Size` field exploits the vulnerability. During exploration 9 new nodes are discovered but only one new path is discovered that leads back to the vulnerability point. For eight of the newly discovered nodes, none of their branches lead to the vulnerability point. The right hand side of Figure 9 shows the complete VPRP returned by the simplifier. As shown in the figure, the successful test checks whether the version field is 0x300 (Windows 3.0). When reverted it checks if the version field is 0x100 (Windows 1.0). Thus, if the version is 0x300 or 0x100 then the vulnerability point is reached, otherwise it is not. Note that blackbox approaches, such as ShieldGen, can only find such conditions by probing all the field values (2^{16} in this case), which they usually avoid to do because it is too expensive. In ShieldGen, after a few failed tests they would likely

```
strcasecmp(METHOD, "get") == 0 ^
URI[0] == '/' ^
URI[1] != '/' ^
strcmp(URI_sub[1], "..") != 0 ^
strncmp(URI_sub[1], "../", 3) != 0 ^
strstr(URI_sub[1], "/../" ) == 0 ^
strcmp(URI_sub[2], "/..") != 0 ^
((isnotdir(URI_sub[1]) ^
  stat(URI_sub[1], ptr) < 0) ||
 (isdir(URI_sub[1]) ^
  stat(URI_sub[1]+"index.html", ptr) < 0) ||
 (URI_sub[1] == 0 ^ stat("index.html", ptr) < 0
) ^
URI[1] != 0 ^
length(URI) > 679
```

Figure 10: Optimal signature for the Atphttpd http_send_error exploit

```
(strcmp(METHOD, "GET") != 0 ^
   length(METHOD) > 165) ||
(strcmp(METHOD, "GET") == 0 ^
   strstr(URI, "/..") != 0 ^
   length(URI) > 170) ||
(strcmp(METHOD, "GET") == 0 ^
   strstr(URI, "/..") == 0 ^
   length(URI) + length(ClientAddr) > 166)
```

Figure 11: Optimal signature for the GHttpd Log exploit. There is a fourth case that requires the attacker to guess configuration information from the server. We omit it for simplicity.

decide that the value of the `Version` field has to be 0x300 for the vulnerability point to be reached, which would allow the attacker to easily avoid detection by changing the value of the field to 0x100.

Windows DCOM RPC. The original protocol-level path-predicate for the DCOM RPC exploit contains 662 conditions, while the message format contains hundreds of fields. During exploration 8 new paths are discovered that lead to the vulnerability point. Figure 8 shows the relevant portion of the protocol-level exploration graph for all the newly discovered paths.

On the left hand side, Figure 8A shows 5 vertically aligned conditions that were added by the original protocol-level path-predicate. The top four conditions depend on the `MaxXmitFrag` (MX) or `MaxRecvFrag` (MR) fields and the last one on the `AuthLength` (AL) field. The remaining 3 nodes in Figure 8A were discovered by the exploration. Interestingly, when extracting the VPRP from the protocol-level exploration graph, the 7 nodes that depend on MX and MR can be removed since both out-edges always point to the same successor. Thus, the whole Figure 8A simplifies to the node `AuthLength == 0`. Since there are no more conditions on the original protocol-level path-predicate that use either the `MaxXmitFrag` or `MaxRecvFrag` fields, that means that the exploration allows us to confidently remove those two fields from the final VPRP.

On the right hand side, Figure 8B shows the other new path to the vulnerability point that was discovered. It was found by exploring the other branch of the `(Flags & 0x80) == 0` condition at the top. The new path traverses a new node and merges back into the original protocol-level path-predicate at a latter condition. The reason behind this new path is that the program is checking if the optional 16-byte `Object` header is present in the RPC message. If it is, then the size of the message is required to be 16 bytes longer (20 rather than 4) to account for it. The final VPRP contains 423 conditions and is too large to be shown here.

7. Conclusion

In this paper we address the main challenge remaining for signature generation: increasing the coverage of vulnerability-based signatures. We have proposed protocol-level constraint-guided exploration, a novel approach to automatically generate high coverage, yet compact, vulnerability point reachability predicates. The three salient properties of our approach are that 1) it is *constraint-guided*, thus much more effective than black-box probing, 2) it works at the *protocol level*, thus the constraints are more resistant to variations in the input such as field reordering or changes in the size of the variable-length fields, and 3) it *merges* paths, which allows to significantly reduce the exploration space and to generate compact vulnerability point reachability predicates. Our experimental results demonstrate that our approach is extremely effective, generating vulnerability point reachability predicates with high coverage (optimal or close to optimal in cases) and small size (often human-readable), offering dramatic improvements over previous approaches. Our work also has applicability in other applications such as patch testing. The efficient exploration may also be of independent interest to other applications requiring the exploration of large program execution space such as automatic test case generation. We hope our work sheds new light on directions for future work.

8. References

[1] A Dumb Patch? http://blogs.technet.com/msrc/archive/2005/10/31/413402.aspx.

[2] QEMU. http://fabrice.bellard.free.fr/qemu/.

[3] SecurityFocus. `http://www.securityfocus.com.`

[4] ShieldGen DCOM RPC Signature. http://research.microsoft.com/research/shield/papers/blasterFilter.htm.

[5] The Metasploit Framework. `http://framework.metasploit.com.`

[6] Wireshark. http://www.wireshark.org.

[7] Witheld for Anonymous Submission.

[8] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques and Tools. Second Edition.* Addison Wesley, 2007.

[9] N. Borisov, D. Brumley, H. J. Wang, J. Dunagan, P. Joshi, and C. Guo. A Generic Application-Level Protocol Analyzer and Its Language. *Network and Distributed System Security Symposium,* San Diego, CA, February 2007.

[10] D. Brumley, J. Newsome, D. Song, H. Wang, and Somesh Jha. Towards Automatic Generation of Vulnerability-Based Signatures. *IEEE Symposium on Security and Privacy,* 2006.

[11] D. Brumley, H. Wang, S. Jha, and D. Song. Creating Vulnerability Signatures Using Weakest Pre-Conditions. *Proceedings of the 2007 Computer Security Foundations Symposium,* Venice, Italy, July 2007.

[12] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis. *ACM Conference on Computer and Communications Security,* Alexandria, VA, October 2007.

[13] C. Cadar, V. Ganesh, P. M. Pawlowski, D. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. *ACM Conference on Computer and Communications Security,* Alexandria, VA, October 2006.

[14] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing Software By Blocking Bad Input. *Symposium on Operating Systems Principles,* Bretton Woods, NH, October 2007.

[15] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-End Containment of Internet Worms. *Symposium on Operating Systems Principles,* Brighton, United Kingdom, October 2005.

[16] W. Cui, M. Peinado, H. J. Wang, and M. Locasto. ShieldGen: Automatic Data Patch Generation for Unknown Vulnerabilities with Informed Probing. *IEEE Symposium on Security and Privacy,* 2007.

[17] E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM,* 18(8), August 1975.

[18] P. Godefroid. Compositional Dynamic Test Generation. *Symposium on Principles of Programming Languages,* Nice, France, January 2007.

[19] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. *SIGPLAN Conference on Programming Language Design and Implementation,* Chicago, IL, June 2005.

[20] H.-A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. *USENIX Security Symposium,* San Diego, CA, August 2004.

[21] C. Kreibich and J. Crowcroft. Honeycomb - Creating Intrusion Detection Signatures Using Honeypots. *Workshop on Hot Topics in Networks,* Boston, MA, November 2003.

[22] Z. Li, M. Sanghi, B. Chavez, Y. Chen, and Ming-Yang Kao. Hamsa: Fast Signature Generation for Zero-Day Polymorphic Worms with Provable Attack Resilience. *IEEE Symposium on Security and Privacy,* 2006.

[23] Z. Liang and R. Sekar. Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers. *ACM Conference on Computer and Communications Security,* Alexandria, VA, November 2005.

[24] Z. Liang and R. Sekar. Automatic Generation of Buffer Overflow Attack Signatures: An Approach Based on Program Behavior Models. *Annual Computer Security Applications Conference,* Tucson, AZ, December 2005.

[25] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. *Network and Distributed System Security Symposium,* 2008.

[26] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. *Network and Distributed System Security Symposium,* San Diego, CA, February 2005.

[27] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. *IEEE Symposium on Security and Privacy,* 2005.

[28] R. Pang, V. Paxson, R. Sommer, and L. Peterson. Binpac: A Yacc for Writing Application Protocol Parsers. *Internet Measurement Conference,* Rio de Janeiro, Brazil, October 2006.

[29] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. *Symposium on Operating System Design and Implementation,* San Francisco, CA, December 2004.

[30] Symantec. Internet Security Threat Report. `http://www.symantec.com/business/theme.jsp?themeid=threatreport,` April 2008.

[31] X. Wang, Z. Li, J. Xu, M. K. Reiter, C. Kil, and J. Y. Choi. Packet Vaccine: Black-Box Exploit Detection and Signature Generation. *ACM Conference on Computer and Communications Security,* Alexandria, VA, October 2006.

[32] G. Wondracek, P. M. Comparetti, C. Kruegel, and E. Kirda. Automatic Network Protocol Analysis. *Network and Distributed System Security Symposium,* 2008.

[33] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt. Automatic Diagnosis and Response to Memory Corruption Vulnerabilities. *ACM Conference on Computer and Communications Security,* Alexandria, VA, November 2005.

[34] V. Yegneswaran, J. T. Giffin, P. Barford, and Somesh Jha. An Architecture for Generating Semantics-Aware Signatures. *USENIX Security Symposium,* 2005.