Towards Generating High Coverage Vulnerability-Based Signatures with Protocol-Level Constraint-Guided Exploration

Juan Caballero[2][1], Zhenkai Liang[3], Pongsin Poosankam[2][1], Dawn Song[1]

[1] UC Berkeley
[2] Carnegie Mellon University
[3] National University of Singapore

Abstract. Signature-based input filtering is an important and widely deployed defense. But current signature generation methods have limited coverage and the generated signatures often 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. In our experimental results, our tool, *Elcano*, generates compact, high coverage signatures for real-world vulnerabilities.

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 [1]. For many of these vulnerabilities, the exploit development time is less than a day, while the patch development time is often days or months [1]. 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 Intrusion Prevention (IPS) and Intrusion Detection (IDS) systems. 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, it is desirable to be able to generate signatures without access to the source code. This is crucial to wide deployment since it enables 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.

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 [2–10]. More recently, researchers proposed to generate *vulnerability-based signatures*, which are generated by analyzing the vulnerable program and its execution and the actual conditions needed to exploit the vulnerability and can guarantee a zero false positive rate [11, 12].

Automatic vulnerability signature generation. 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* (VPRP). 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. While both problems are important, the space limitations makes trying to cover both in a single paper unrealistic. Thus, in this paper we focus on how to generate vulnerability point reachability predicates with high coverage and compact size, and we refer the reader to [13] for details on the vulnerability condition extraction. In this paper, we use *optimal signature* to refer to a vulnerability signature that has no false positives and no false negatives.

Coverage is a key challenge. One important problem with early vulnerability-based signature generation approaches [11, 12] is that the signatures only capture a single path to the vulnerability point (i.e., their VPRP 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 [14, 15].

Acknowledging the importance of enhancing the coverage of vulnerability-based signatures, recent work tries to incorporate multiple paths into the VPRP either by static analysis [16], or by dynamic analysis [17, 18]. However, performing precise static analysis on binaries is hard due to issues such as indirection, pointers and loops.

ShieldGen takes a probing-based approach using protocol format information [18]—using the given protocol format, it generates different well-formed variants of the original exploit using various heuristics and then checks whether any of the variants still exploits the vulnerability. The advantage of this approach is that by using protocol format information, the final signature is expressed at the protocol level (which we call *protocol-level* signature) instead of the byte level. Compared to signatures at the byte-level (which do not understand the protocol format), protocol-level signatures have two advantages: they are more compact and they naturally cover variants of the exploits caused by variable-length fields and field re-ordering (See more detail in Section 2.2). The disadvantage of the approach used by ShieldGen is that the exploration uses heuristics to figure out what test inputs to generate. Such heuristics can introduce false positives and do not use the information from the execution of the program, which would increase the coverage of the program execution space. As a result, the exploration is inefficient and has various limitations (See Section 2.3).

Bouncer extends previous approaches using symbolic execution to generate symbolic constraints on inputs as signatures [17]. Even though Bouncer makes improvements in increasing the coverage of the generated signatures, it still suffers from several limitations. First, it generates byte-level signatures instead of protocol-level signatures. As a result, it is difficult for Bouncer to handle evasion attacks using variable-length fields and field re-ordering. Second, Bouncer's exploration is inefficient and largely heuristic-based. As mentioned in their paper, the authors tried to use symbolic-constraint-guided exploration to explore the program execution space to identify different paths reaching the vulnerability point, but couldn't make the approach scale to real-world programs and thus had to resort to 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. Thus, a key open problem for generating accurate and efficient signatures is how to generate vulnerability point reachability predicates with high coverage.

Our approach. In this paper, we propose *protocol-level constraint-guided exploration*, a new approach to automatically generate vulnerability point reachability predicates with high coverage, 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 heuristics-based exploration as in ShieldGen and Bouncer), 2) the constraint-guided exploration works at the *protocol-level* and generates protocol-level signatures at the end, and 3) it effectively *merges* explored execution paths to remove redundant exploration. The three points seamlessly weave together and amplify each other's benefit. By using constraint-guided exploration, our approach significantly increases the effectiveness and efficiency of the program execution space exploration. By lifting the symbolic constraints from the byte level to the protocol level, our constraint-guided exploration is done at the protocol level, which makes the exploration feasible for real-world programs, addressing the problem that Bouncer couldn't solve. By merging paths in the exploration, we further reduce the exploration space.

Elcano. We have designed and developed *Elcano*, realizing the aforementioned approach. We have evaluated the effectiveness of our system using real-world vulnerable programs. In our experiments, Elcano achieved optimal or close-to-optimal results in terms of coverage. In addition, the generated signatures are compact. In fact, most of the signatures are so compact that they can be understood by a human.

Compared to Bouncer, Elcano produces higher coverage signatures. For example, for the GHttpd vulnerability Bouncer run for 24 hours, exploring only some fraction of all possible paths, and produced a partial signature with significant false negatives. In contrast, Elcano generates an optimal signature for the same vulnerability in 55 seconds. Compared to ShieldGen, Elcano produces more accurate signatures, both in terms of less false negatives (i.e., higher coverage) and less false positives.

In addition to signature generation, extracting a high coverage vulnerability point reachability predicate is useful for other applications such as exploit generation [19] and 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 [20]. This situation is not uncommon. A quick search on the CVE database returns 13 vulnerabilities that were incorrectly or incompletely patched [21]. Our technique could assist software developers to build more accurate patches. Furthermore, our protocol-level constraint-guided approach can increase the effectiveness of generating high-coverage test cases and hence be very valuable to software testing and bug finding.

2 Problem Definition and Approach Overview

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

2.1 Problem Definition

Automatic generation of protocol-level vulnerability point reachability predicates. 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.

Parser availability and specification quality. The problem of automatic generation of protocol-level vulnerability point reachability predicates assumes the availability of a parser implementing a given protocol or file specification. Such requirement is identical to previous approaches such as ShieldGen [18]. The parser given some input data can map it into fields, according to the specification, or fail if the input is malformed. In the latter case, the IDS/IPS could opt to block the input or let it go through while logging the event or sending a warning. Such parser is available for common protocols (e.g., Wireshark [22]), and many commercial network-based IDS or IPS have such a parser built-in. In addition, recent work has shown how to create a generic parser that takes as input multiple protocol specifications written in an intermediate language [23, 24].

The quality of the specification used by the parser matters. While obtaining a high quality specification is not easy, this is a one time effort, which can be reused for multiple signatures, as well as other applications. For example, in our experiments we extracted a WMF file format specification. According to the CVE Database [21] the WMF file format appears in 21 vulnerabilities, where our specification could be reused. Similarly, an HTTP specification could be reused in over 1500 vulnerabilities. Also, recent work has proposed to automatically extract the protocol specification from the program binary [25–28]. Such work can be used when the protocol used by the vulnerable program has no public specification.

Exploit availability. Similarly to all previous work on automatic generation of vulnerability signatures [11, 12, 17, 18], our problem definition assumes that an initial exploit message is given.

```
1  void service() {
2   char msgBuf[4096];
3   char lineBuf[4096];
4   int nb=0, i=0, sockfd=0;
5   nb=recv(sockfd,msgBuf,4096,0);
6   for(i = 0; i < nb; i++) {
7     if (msgBuf[i] == ’\n’)
8         break;
9     else
10      lineBuf[i] = msgBuf[i];
11  }
12  if (lineBuf[i-1] == ’\r’)
13    lineBuf[i-1] = ’\0’
14  else lineBuf[i] = ’\0’;
15  doRequest(lineBuf);
16 }
17 void doRequest(char *lineBuf){
18  char vulBuf[128],uri[256];
19  char ver[256], method[256];
20  int is_cgi = 0;
21  sscanf(lineBuf,
22    "%255s %255s %255s",
23    method, uri, ver);
24  if (strcmp(method,"GET")==0 ||
25      strcmp(method,"HEAD")==0){
26    if strncmp(uri,"/cgi-bin/",
27      9)==0  is_cgi = 1;
28    else is_cgi = 0;
29    if (uri[0] != ’/’) return;
30    strcpy(vulBuf, uri);
31  }
32 }
```

Fig. 1. Our running example.

Vulnerability point availability. Finally, our problem definition assumes that the vulnerability point is given. Identifying the vulnerability point is part of a parallel project that aims to accurately describe the vulnerability condition [13]. Such vulnerability point could also be identified using previous techniques [17, 29].

2.2 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 example, the `service` function copies one line of data received over the network into `linebuf` and passes it to the `doRequest` function that parses it into several field variables (lines 21-23) and performs some checks on the field values (lines 24-31). 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 24-25), and the first character of the URI is a slash (line 29), then the vulnerability point is reached at line 30, where the size of `vulBuf` is not checked by the `strcpy` function. Thus, a long URI can overflow the `vulBuf` buffer.

In this example, the vulnerability point is at line 30, and the vulnerability condition is that the local variable `vulBuf` will be overflowed if the size of the URI field in the received message is greater than 127. Therefore, for this example, the vulnerability point reachability predicate is: `(strcmp(FIELD_METHOD,"GET") == 0 || strcmp(FIELD_METHOD,"HEAD") == 0) && FIELD_URI[0]` $\neq$ `’/’` while the vulnerability condition is: `length(FIELD_URI) > 127`, and the conjunction of the two is an optimal protocol-level signature.

2.3 Approach

In this paper we propose a new approach to generate high coverage, yet compact, vulnerability point reachability predicates, called *protocol-level constraint-guided explo-*

ration. Next, we give the motivation and an overview of the three characteristics that comprise our approach.

Constraint-guided. As mentioned in Section 1, previous approaches such as ShieldGen and Bouncer use heuristics-based exploration [17,18]. Heuristic-based exploration suffers from a fundamental limitation: the number of probes needed to exhaustively search the whole space is usually astronomical. In addition, an exhaustive search is inefficient as many probes end up executing the same path in the program. Thus, such approaches often rely on heuristics that are not guaranteed to significantly increase the signature's coverage and can also introduce false positives.

For example, ShieldGen [18] first assumes that fields can be probed independently, and then for fixed-length fields it samples just a few values of each field, checking whether the vulnerability point is reached or not for those values. Probing each field independently means that conditions involving multiple fields cannot be found. Take the condition `SIZE1 + SIZE2` $\leq$ `MSG_SIZE`, where `SIZE1` and `SIZE2` are length fields in the input, and `MSG_SIZE` represents the total length of the received message. The authors of ShieldGen acknowledge that their signatures cannot capture this type of conditions, but such conditions are commonly used by programs to verify that the input message is well-formed and failing to identify them will introduce either false positives or false negatives, depending on the particular heuristic. Probing only a few sample values for each field is likely to miss constraints that are satisfied by only a small fraction of the field values. For example, a conditional statement such as `if (FIELD==10) || (FIELD==20) then exploit, else safe`, where FIELD is a 32-bit integer, creates two paths to the vulnerability point. Finding each of these paths would require 2^{30} random probes on average to discover. Creating a signature that covers both paths is critical since if the signature only covers one path (e.g., `FIELD == 10`), the attacker could easily evade detection by changing FIELD to have value 20.

To overcome these limitations, 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. This constraint-guided exploration is similar in spirit to recent work on using symbolic execution for automatic test case generation [30–32]. However, simply applying those techniques does not scale to real-world programs, given the exponential number of paths to explore. In fact, in Bouncer [17] 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 heuristics-based probing approach.

To make the constraint-guided exploration feasible and effective we have incorporated two other key characteristics into our approach as described below.

Protocol-level constraints. 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 [6,18]. We call such signatures *protocol-level signatures*.

Our contribution here is to show that, by lifting stream-level conditions to *protocol-level conditions*, so that they operate on protocol fields rather than on the input bytes, we can make the constraint-guided approach feasible, as using constraints at the protocol-level hugely reduces the number of paths to be explored compared to using stream-level conditions. The state reduction is achieved in two ways. First, the parsing logic often introduces huge complexity in terms of the number of execution paths that need to be analyzed. For example, in our experiments, 99.8% of all constraints in the HTTP vulnerabilities are generated by the parsing logic. While such parsing constraints need to be present in the stream-level conditions, they can be removed in the protocol-level conditions. Second, the stream-level conditions 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). Unless the parsing conditions are removed the resulting signature would be very easy to evade by an attacker by applying small variations to the field structure of the exploit message. Finally, the vulnerability point reachability predicates at the protocol level are smaller and easier to understand by humans.

Merging execution paths. The combination of protocol-level conditions with constraint-guided exploration is what we call *protocol-level constraint-guided exploration*, an iterative process that incrementally discovers new paths leading to the vulnerability point. Those paths need to be added to the vulnerability point reachability predicate. The simplistic approach would be to blindly explore new paths by reversing conditions and at the end create a vulnerability point reachability predicate that is a disjunction (i.e., an enumeration) of all the discovered paths leading to the vulnerability point. Such approach has two main problems. First, blindly reversing conditions produces a search space explosion, since the number of paths to explore becomes exponential in the number of conditions, and much larger than the real number of paths that exist in the program. We explain this in detail in Section 4. In addition, merely enumerating the discovered paths generates signatures that quickly explode in size.

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 HTTP URL 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. This helps alleviate the search space explosion problem, and allows our exploration to quickly reach high coverage.

2.4 Architecture Overview

We have implemented our approach in a system called Elcano. The architecture of Elcano is shown in Figure 2. It comprises of two main components: the *constraint extractor* and the *exploration module*, and two off-the-shelf assisting components: the *execution monitor* and *the parser*.

The overall exploration process is an iterative process that incrementally explores new execution paths. In each iteration (that we also call test), an input is sent to the program under analysis, running inside the execution monitor. The execution monitor

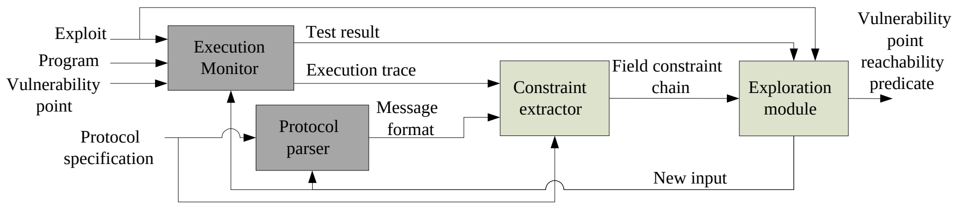


Fig. 2. Elcano architecture overview. The darker color modules are given, while the lighter color components have been designed and implemented in this work.

produces an execution trace that captures the complete execution of the program on the given input, including which instructions were executed and the operands content. The execution monitor also logs the test result, i.e., whether the vulnerability point was reached or not during the execution. In addition, the parser extracts the message format for the input, according to the given protocol specification.

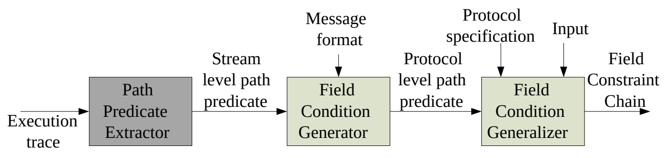
Then, given the execution trace and the message format, the constraint extractor obtains the *field constraint chain*. The field constraint chain is conceptually similar to the *path predicate* used in previous work on automatic test case generation, but the conditions are at the protocol-level and each condition is tagged with additional information. We detail the field constraint chain and its construction in Section 3.

The exploration module maintains the *protocol-level exploration graph*, which stores the current state of the exploration, i.e., all the execution paths that have been so far explored. Given the field constraint chain, the exploit message and the test result, the exploration module merges the new field constraint chain into the current protocol-level exploration graph. Then, the exploration module uses the protocol-level exploration graph to select a new path to be explored and generates a new input that will lead the program execution to traverse that path. Given the newly generated input, another iteration begins. We detail the exploration module in Section 4.

The process is started with the initial exploit message and runs iteratively until there are no more paths to explore or a user-specified time-limit is reached. At that point the exploration module outputs the VPRP. The VPRPs produced by Elcano are written using the Vine language [33] with some extensions for string operations [34]. The Vine language is part of the Bitblaze binary analysis infrastructure [35].

3 Extracting the Protocol-Level Path-Predicate

In this section we present the constraint extractor, which given an execution trace, produces a field constraint chain. The architecture of the constraint extractor is shown in Figure 3. First, given the execution trace the *path predicate extractor* performs symbolic execution with the input represented as a symbolic variable and extracts the *path predicate*, which is essentially the conjunction of all branch conditions dependent on the symbolic input in the execution captured in the execution trace. The concept of symbolic execution, the path predicate and how to compute it are well understood and have been widely used in previous work including vulnerability signature generation [11,12] and automatic test case generation [30,31]. Thus, we refer the interested reader to these previous work for details.



Execution trace
Path Predicate Extractor
Stream level path predicate
Message format
Field Condition Generator
Protocol level path predicate
Protocol specification
Input
Field Condition Generalizer
Field Constraint Chain

Fig. 3. Constraint Extractor Architecture. The darker color module is given, while the lighter color components have been designed and implemented in this work.

The path predicate generated by previous work is at the stream-level, i.e., the conditions are on raw bytes of the input. To enable constraint-guided exploration, Elcano needs to lift the path predicate from the stream-level to the protocol-level, where the conditions are instead on field variables of the input. 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*. In addition, the constraint extractor needs to remove the parsing conditions, which dramatically reduces the exploration space and makes the constraint-guided exploration feasible.

To accomplish this, first the *field condition generator* lifts the stream-level path-predicate to the protocol-level, and then the *field condition generalizer* generalizes it by removing the parsing conditions and outputs the *field constraint chain*, which is essentially the protocol-level path-predicate, where each condition is annotated with some additional information and conditions are ordered using the same order as they appeared in the execution.

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.

The message format provided by the parser is a hierarchical tree, where one field may contain different subfields, with the root of the tree representing the whole message. For example, the `linebuf` variable in our running example represents the `Request-Line` field, which in turn contains 3 subfields: `Method`, `Request-URI`, and `HTTP-Version`. Thus, a condition such as: `strstr(linebuf,"../")` $\neq$ `0` would be translated as `strstr(FIELD_Request-Line,"../")` $\neq$ `0`. A condition on the whole message would translate into a condition on the special `MSG` field.

Benefits. This step lifts the stream-level path-predicate to the protocol-level, breaking 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 signatures. Thus, by expressing constraints using field symbols, protocol-level signatures 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, the protocol specification and the input that was sent to the program and outputs a field constraint chain where the parsing-related conditions have been removed.

First, the field condition generalizer assigns a symbolic variable to each byte of the input and processes the input according to the given protocol specification. This step generates symbolic conditions that capture the constraints on the input which restrict the message format of the input to be the same as the message format returned by the parser on the given input. We term these conditions the parsing conditions. Then, the field condition generalizer removes the parsing conditions from the protocol-level path-predicate by using a fast syntactic equivalence check. If the fast syntactic check fails, the field condition generalizer uses a more expensive equivalence check that uses a decision procedure.

Benefits. The parsing conditions in the protocol-level path-predicate over-constrain the variable-length fields, forcing them to have some specific size (e.g., the same as in the exploit message). Thus, removing the parsing conditions allows the vulnerability point reachability predicate to handle exploit messages where the variable-length fields have a size different than in the original exploit message. 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. Note that the parsing conditions are enforced by the parser, so we can safely remove them from the protocol-level path-predicate while still having the conditions enforced during the signature matching time. We refer the reader to the extended version for more details [36].

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 generalized protocol-level path-predicate (after the parsing conditions have been removed). A field constraint chain is an enhanced version of the protocol-level path-predicate where each branch condition is annotated with the instruction counter and an MD5 hash of the callstack of the program at the 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, which adds the given field constraint chain to the protocol-level exploration graph, selects a new path to be explored and generates an input that will traverse that path. That input is used to start a new iteration of the whole process by sending it to the program running in the execution monitor. Once there are no more paths to explore or a user-specified time-limit is reached, the exploration module stops the exploration and outputs the VPRP.

Our exploration is based on a *protocol-level exploration graph*, which makes it significantly different from the traditional constraint-based exploration used in automatic test case generation approaches [30, 31, 37]. Using a protocol-level exploration graph provides two fundamental benefits: 1) the exploration space is significantly reduced, and 2) it becomes easy to merge paths, which in turn further reduces the exploration space, and reduces the size of the vulnerability point reachability predicate. In this section, we first introduce the protocol-level exploration graph, next we present our intuition for merging paths, and then we describe the exploration process used to extract the vulnerability point reachability predicate.

4.1 The Protocol-Level Exploration Graph

The explorer dynamically builds a *protocol-level exploration graph* as the exploration progresses. In the graph, each node represents an input-dependant branching point (i.e., a conditional jump) in the execution, which comprises the protocol-level condition and some additional information about the state of the program when the branching point was reached, which we explain in Section 4.2. 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.

4.2 Merging Execution Paths

When a new field constraint chain is added to the protocol-level exploration graph, it is important to merge all conditions in the field constraint chain that are already present in the graph. Failure to merge a condition creates a duplicate node, which in turn effectively doubles the exploration space because all the subtree hanging from the replicated node would need to be explored as well. Thus, as the number of duplicated nodes increases, the exploration space increases exponentially.

The key intuition behind why merging is necessary 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

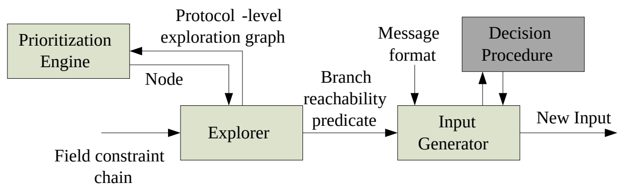


Fig. 4. Exploration module architecture. The darker color module is given, while the lighter color components have been designed and implemented in this work.

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.

The intuition on the merging is that two nodes can be merged if they represent the same program point and they are reached with the same program state. To identify the program point, each condition in the field constraint chain is annotated with the program's instruction counter (*eip*) and an MD5 hash of the callstack, both taken at the time the condition was executed. To identify the program state we use a technique similar to the one introduced in [38] where we compute the set of all values (both concrete and symbolic) written by the program during the execution up to the point where the condition is executed. Thus, we merge nodes that satisfy 4 conditions: same eip, same callstack hash, equivalent conditions, and same program state, where Elcano queries the decision procedure to determine if two conditions are equivalent.

4.3 The Exploration Process

Figure 4 shows the architecture of the exploration module. It is comprised of three components: the *explorer*, the *prioritization engine*, and the *input generator*, plus an off-the-shelf *decision procedure*. The exploration process is comprised of 3 steps: (1) given the field constraint chain, the explorer adds it to the current protocol-level exploration graph producing an updated graph; (2) given the updated protocol-level exploration graph, the prioritization engine decides which node's open edge to explore next; (3) 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.

The new input is then used to start another iteration of the whole process as shown in Figure 2, that is, the new input is replayed to the program running in the execution monitor and a new field constraint chain is generated by the constraint extractor, which is passed to the explorer and so on. The prioritization engine is in charge of stopping the whole process once there are no more paths to explore or a user-specified time-limit is reached. When the exploration stops, the explorer outputs the VPRP.

Next, we detail the 3 steps in the exploration process and how to extract the VPRP. We illustrate the different steps using Figure 5 which represents the graph for our run-

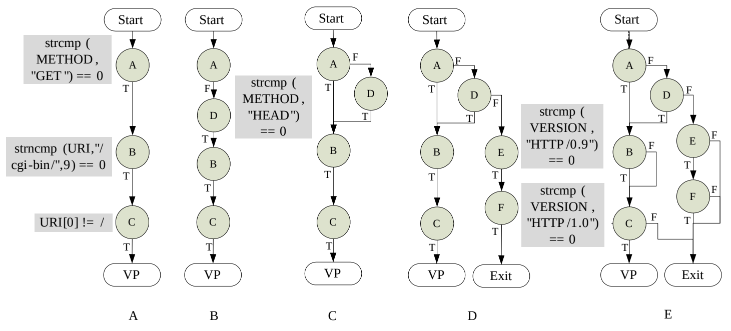

Start
strcmp (METHOD , "GET ") == 0
strncmp (URI,"/ cgi-bin/",9) == 0
URI[0] != /
strcmp (METHOD , "HEAD") == 0
strcmp (VERSION , "HTTP/0.9") == 0
strcmp (VERSION , "HTTP/1.0") == 0
VP
Exit
A
B
C
D
E

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

ning 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.

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 top until it finds a node that it cannot merge, either because it is not in the graph yet, or because the successor in the new field constraint chain is not the same one as in the graph. To check if the node is already in the graph, the explorer checks if the node to be inserted is equivalent (same EIP, same callstack hash, equivalent condition, and same state) to any other node already in the graph. We call the last node that can be merged from the top the *split node*.

Once a split node has been identified the explorer keeps trying to merge the rest of the nodes in the new field constraint chain until it finds a node that it can merge, which we term the *join node*. At that point, the explorer adds all the nodes in the new field constraint chain between the split node and the join node as a sequence of nodes in the graph hanging from the split node and merging at the join node. The process of looking for a split node and then for a join node is repeated until the sink of the new field constraint chain is reached. At that point, if the explorer was looking for a join node then all nodes between the last split node and the sink are added to the graph as a sequence that hangs from the last split node and ends at the sink.

For example, in Figure 5A the graph contains only the original field constraint chain generated by sending the starting exploit message to the program, which contains the three nodes introduced by lines 24, 26, and 29 in our running example (since the parsing conditions have already been removed). The sink of the original field constraint chain is the vulnerability point node (VP). Figure 5B shows the second field constraint chain that is added to the graph, which was obtained by creating an input that traverses the false branch of node A. When adding the field constraint chain in Figure 5B to the graph in Figure 5A, the explorer merges node A and determines that A is a split node because A's successor in the new field constraint chain is not A's successor in the graph. Then, at node B the explorer finds a join node and adds node D between the split node and the join node in the graph. Finally node C is merged and we show the updated graph in Figure 5C.

Selecting the next node to explore. Even after removing the parsing conditions and merging nodes, the number of paths to explore can still be large. Since we are only interested in paths that reach the vulnerability point, we have implemented a simple prioritization scheme that favours paths that are more likely to reach it. The prioritization engine uses a simple weight scheme, where there are three weights 0, 1, and 2. Each weight has its own node queue and the prioritization engine always picks the first node from the highest weight non-empty queue. The explorer assigns the weights to the nodes when adding them to the graph. Nodes that represent loop exit conditions get a zero weight (i.e., lowest priority). Nodes in a field constraint chain that has the VP as sink get a weight of 2 (i.e., highest priority). All other nodes get a weight of 1. We favor nodes that are in a path to the VP because 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 less likely to reach VP later. We disfavor loop exit conditions to delay unrolling the same loop multiple times. Such heuristic helps achieve high coverage quickly.

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, which includes all paths in the graph from the Start to that node. Similarly, we define a *branch reachability predicate* to be the predicate that summarizes how to traverse a specific branch of a node. A branch reachability predicate is the conjunction of a node reachability predicate with the node's condition (to traverse the true branch), or the negation of the node's condition (to traverse the false branch). To compute a new input that traverses the specific branch 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) [39] 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 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 5C 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: $\overline{A}$ `&&` $\overline{D}$. Similarly, in Figure 5D 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`||(`$\overline{A}$ `&&` D`))` `&&` $\overline{B}$.

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 input can reach that branch, then the branch is connected to the `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

Program	CVE	Protocol	Type	Guest OS	Vulnerability Type
gdi32.dll (v3159)	CVE-2008-1087	EMF file	Binary	Windows XP	Buffer overflow
gdi32.dll (v3099)	CVE-2007-3034	WMF file	Binary	Windows XP	Integer overflow
Windows DCOM RPC	CVE-2003-0352	RPC	Binary	Windows XP	Buffer overflow
GHttpd	CVE-2002-1904	HTTP	Text	Red Hat 7.3	Buffer overflow
AtpHttpd	CVE-2002-1816	HTTP	Text	Red Hat 7.3	Buffer overflow
Microsoft SQL Server	CVE-2002-0649	Proprietary	Binary	Windows 2000	Buffer overflow

Table 1. Vulnerable programs used in the evaluation.

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. We refer the reader to our extended version [36] for our handling of field conditions that depend on a memory read from a symbolic address.

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 VPRP from the graph the explorer computes the node reachability predicate for the VP node. For our running example, represented in Figure 5E the VPRP is: `(`A`||(`$\overline{A}$ `&&` D`))` `&&` C. Note that, a mere disjunction of all paths to the VP, would generate the following VPRP: `(`A `&&` B `&&` C`)||(`$\overline{A}$ `&&` D `&&` B `&&` C`)||(`A `&&` $\overline{B}$ `&&` C`)||(`$\overline{A}$ `&&` D `&&` $\overline{B}$ `&&` C`)`. Thus, Elcano's VPRP is more compact using 4 conditions instead of 14.

5 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 Elcano using 6 vulnerable programs, summarized in Table 1. The table shows the program, the CVE identifier for the vulnerability [21], the protocol used by the vulnerable program, the protocol type (i.e., binary or text), the guest operating system used to run the vulnerable program, and the type of vulnerability. We select the vulnerabilities to cover file formats as well as network protocols, multiple operating systems, multiple vulnerability types, and both open-source and closed programs, where no source code is available. In addition, the older vulnerabilities (i.e., last four) are also selected because they have been analyzed in previous work, and this allows us to compare our system's results to previous ones.

5.1 Constraint Extractor Results

In this section we evaluate the effectiveness of the constraint extractor, in particular of the field condition generalizer, at removing the parsing conditions from the protocol-level path-predicate. For simplicity, we only show the results for the protocol-level path-predicate produced by the field condition generator from the execution trace generated by the original exploit. Note that, during exploration this process is repeated once per newly generated input. Table 2 summarizes the results. The *Original* column represents the number of input-dependent conditions in the protocol-level path-predicate and is used as the base for comparison. The *Non-parsing conditions* column shows the number of remaining conditions after removing the parsing conditions.

Program	Original	Non-parsing conditions
Gdi-emf	860	65
Gdi-wmf	4	4
DCOM RPC	535	521
GHttpd	2498	5
AtpHttpd	6034	10
SQL Server	2447	7

Table 2. Constraint extractor results for the first test, including the number of conditions in the protocol-level path-predicate and the number of remaining conditions after parsing conditions have been removed.

Program	All branches explored	VPRP
Gdi-emf	no	72
Gdi-wmf	yes	5
DCOM RPC	no	1651
GHttpd	yes	3
AtpHttpd	yes	10
SQL Server	yes	3

Table 3. Exploration results, including whether all open edges in the protocol-level exploration graph were explored and the number of conditions remaining in the vulnerability point reachability predicate.

The removal of the parsing conditions is very successful in all experiments. Overall, in the four vulnerable programs that include variable-length strings (i.e., excluding Gdi-wmf and DCOM-RPC), the parsing conditions account for 92.4% to 99.8% of all conditions. For formats that include arrays, such as DCOM RPC, the number of parsing conditions is much smaller but it is important to remove such conditions because otherwise they constrain the array to have the same number of elements as in the exploit message. By removing the parsing conditions, each field constraint chain represents many program execution paths produced by modifying the format of the exploit message (e.g., extending variable-length fields or reordering fields). This dramatically decreases the exploration space making the constraint-guided exploration feasible.

5.2 Exploration Results

Table 3 shows the results for the exploration phase. We set a user-defined time-limit of 6 hours for the exploration. If the exploration has not completed by that time Elcano outputs the intermediate VPRP and stores the current state of the exploration. This state can later be loaded to continue the exploration at the same point where it was interrupted. The first column indicates whether the exploration completes before the specified time-limit. The second column presents the number of conditions in the intermediate VPRP that is output by the exploration module once there are no more paths to be explored or the time-limit is reached.

The results show that in 4 out of 6 experiments Elcano explored all possible paths, thus generating a complete VPRP. For the DCOM RPC and Gdi-emf experiments, the 6 hour time-limit was reached, thus the VPRPs are not complete. They also show that the number of conditions in the VPRP is in most cases small. The small number of conditions in the VPRP and the fact that in many cases those conditions are small themselves, makes the signatures easy for humans to analyze, as opposed to previous constraint-based approaches where the large number of conditions in the signature made it hard to gain insight on the quality of the signature. We do that by labeling the nodes in the graph with the full protocol-level conditions.

Performance. Table 4 summarizes the performance measurements for Elcano. All measurements were taken on a desktop computer with a 2.4GHz Intel Core2 Duo CPU and 4 GB of memory. The first column presents the VPRP generation time in seconds. For

	Gener.		Ave. test	Trace
Program	**time**	**# tests**	**time**	**size**
Gdi-emf	21600	502	43.0	28.8
Gdi-wmf	98	6	16.3	3.0
DCOM RPC	21600	235	92.0	3.5
GHttpd	55	6	9.1	3.0
AtpHttpd	282	12	23.5	8.6
SQL Server	1384	11	125.8	27.5

Table 4. Performance evaluation. The generation time and the average test time are given in seconds, and the trace size is given in Megabytes.

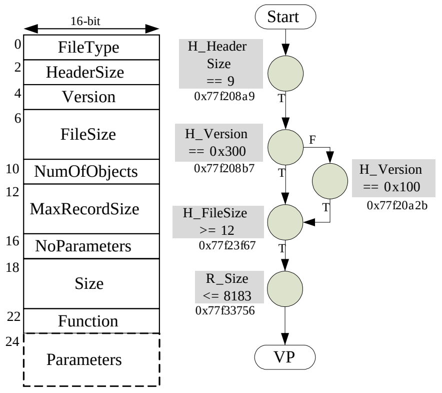


Table 5. On the left, the format of the Gdi-wmf exploit file. On the right the vulnerability point reachability predicate.

the Gdi-emf and DCOM RPC examples, the 6 hour time-limit on generation time is reached. For the rest, the generation time ranges from under one minute for the GHttpd vulnerability up to 23 minutes for the Microsoft SQL vulnerability. Most of the time (between 60% and 80% depending on the example) is spent by the constraint extractor. Thus, we plan to parallelize the exploration by having a central explorer, which spawns multiple copies of the constraint extractor and the execution monitor, each testing a different input and reporting back to the explorer. The remaining columns show the number of tests in the exploration, the average time per test in seconds, and the average size in Megabytes of the execution trace.

Compared to Bouncer, where the authors also analyze the SQL Server and GHttpd vulnerabilities, the signatures produced by Elcano have higher coverage (i.e., less false negatives) and are smaller. For example, Bouncer spends 4.7 hours to generate a signature for the SQL Server vulnerability, and the generated signature only covers a fraction of all the paths to the vulnerability point. In contrast, Elcano spends only 23 minutes, and the generated signature covers all input-dependnt branches to the vulnerability point. Similarly, for the GHttpd vulnerability the authors stop the signature generation after 24 hours, and again the signature only covers a fraction of all input-dependent branches to the vulnerability point, while Elcano generates a complete signature that covers all input-dependent branches to the vulnerability point in under one minute.

SQL server. The parser returns that there are two fields in the exploit message: the Command (CMD) and the Database name (DB). The original protocol-level path-predicate returned by the constraint extractor contains 7 conditions: 4 on the CMD field and the other 3 on the DB field. The exploration explores the open edges of those 7 nodes and finds that none of the newly generated inputs reaches the vulnerability point. Thus, no new paths are added to the graph and the 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` . According to the ShieldGen authors, who had access to the source code, this signature would be optimal.

Gdi-wmf. Figure 5 shows on the left the field structure for the exploit file and on the right the VPRP. The original protocol-level path-predicate contained the 4 aligned nodes on the left of the graph, while the exploration discovers one new path leading to the vulnerability point that introduces the node on the right. The graph shows that the program checks whether the `Version` field is 0x300 (Windows 3.0) or 0x100 (Windows 1.0). Such constraint is unlikely to be detected by probing approaches, since they usually sample only a few values. In fact, in ShieldGen they analyze a different vulnerability in the same library but run across the same constraint. The authors acknowledge that they miss the second condition of the disjunction. Thus, an attacker could easily avoid detection by changing the value of the Version field. Since we have no access to the source we cannot verify if our VPRP is optimal, though we believe it to be.

Other experiments. Due to space constraints we refer the reader to our extended version [36] for details on the Atphttpd, GHttpd and DCOM RPC examples. For the Atphttpd and GHttpd vulnerabilities, where we have access to the source code, the extended version contains the optimal signatures that we manually extracted for the vulnerability. The results show that Elcano's VPRPs exactly match or are very close to the optimal ones that we manually extracted from the source code.

6 Conclusion

In this paper we propose protocol-level constraint-guided exploration, a novel approach to automatically generate high coverage, yet compact, vulnerability point reachability predicates, with application to signature generation, exploit generation and patch verification. Our experimental results demonstrate that our approach is effective, generates small vulnerability point reachability predicates with high coverage (optimal or close to optimal in cases), and offers significant improvements over previous approaches.

7 Acknowledgements

We would like to thank James Newsome and Prateek Saxena for many helpful discussions on signature generation. We also thank Stephen McCamant and the anonymous reviewers for their insightful comments on this document.

This material is based upon work partially supported by the National Science Foundation under Grants No. 0311808, No. 0448452, No. 0627511, and CCF-0424422, and by the Air Force Office of Scientific Research under MURI Grant No. 22178970-4170. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Air Force Office of Scientific Research, or the National Science Foundation.

References

1. Symantec: Internet security threat report. `http://www.symantec.com/business/theme.jsp?themeid=threatreport` (2008)
2. Kreibich, C., Crowcroft, J.: Honeycomb - creating intrusion detection signatures using honeypots. In: Workshop on Hot Topics in Networks, Boston, MA (2003)

3. Kim, H.A., Karp, B.: Autograph: Toward automated, distributed worm signature detection. In: USENIX Security Symposium, San Diego, CA (2004)
4. Singh, S., Estan, C., Varghese, G., Savage, S.: Automated worm fingerprinting. In: Symposium on Operating System Design and Implementation, San Francisco, CA (2004)
5. Newsome, J., Karp, B., Song, D.: Polygraph: Automatically generating signatures for polymorphic worms. In: IEEE Symposium on Security and Privacy, Oakland, CA (2005)
6. Yegneswaran, V., Giffin, J.T., Barford, P., Jha, S.: An architecture for generating semantics-aware signatures. In: USENIX Security Symposium, Baltimore, MD (2005)
7. Li, Z., Sanghi, M., Chen, Y., Kao, M.Y., Chavez, B.: Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In: IEEE Symposium on Security and Privacy, Oakland, CA (2006)
8. Liang, Z., Sekar, R.: Fast and automated generation of attack signatures: A basis for building self-protecting servers. In: ACM Conference on Computer and Communications Security, Alexandria, VA (2005)
9. Liang, Z., Sekar, R.: Automatic generation of buffer overflow attack signatures: An approach based on program behavior models. In: Annual Computer Security Applications Conference, Tucson, AZ (2005)
10. Wang, X., Li, Z., Xu, J., Reiter, M.K., Kil, C., Choi, J.Y.: Packet vaccine: Black-box exploit detection and signature generation. In: ACM Conference on Computer and Communications Security, Alexandria, VA (2006)
11. Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., Barham, P.: Vigilante: End-to-end containment of internet worms. In: Symposium on Operating Systems Principles, Brighton, United Kingdom (2005)
12. Brumley, D., Newsome, J., Song, D., Wang, H., Jha, S.: Towards automatic generation of vulnerability-based signatures. In: IEEE Symposium on Security and Privacy, Oakland, CA (2006)
13. Saxena, P., Poosankam, P., McCamant, S., Song, D.: Loop-extended symbolic execution on binary programs. In: International Symposium on Software Testing and Analysis, Chicago, IL (2009)
14. Vigna, G., Robertson, W., Balzarotti, D.: Testing network-based intrusion detection signatures using mutant exploits. In: ACM Conference on Computer and Communications Security, Washington, D.C. (2004)
15. Rubin, S., Jha, S., Miller, B.P.: Automatic generation and analysis of nids attacks. In: Annual Computer Security Applications Conference, Tucson, AZ (2004)
16. Brumley, D., Wang, H., Jha, S., Song, D.: Creating vulnerability signatures using weakest pre-conditions. In: Computer Security Foundations Symposium, Venice, Italy (2007)
17. Costa, M., Castro, M., Zhou, L., Zhang, L., Peinado, M.: Bouncer: Securing software by blocking bad input. In: Symposium on Operating Systems Principles, Bretton Woods, NH (2007)
18. Cui, W., Peinado, M., Wang, H.J., Locasto, M.: Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In: IEEE Symposium on Security and Privacy, Oakland, CA (2007)
19. Brumley, D., Poosankam, P., Song, D., Zheng, J.: Automatic patch-based exploit generation is possible: Techniques and implications. In: IEEE Symposium on Security and Privacy, Oakland, CA (2008)
20. : A dumb patch? `http://blogs.technet.com/msrc/archive/2005/10/31/413402.aspx`.

21. : Common vulnerabilities and exposures (cve) `http://cve.mitre.org/cve/`.
22. : Wireshark `http://www.wireshark.org`.
23. Pang, R., Paxson, V., Sommer, R., Peterson, L.: Binpac: A yacc for writing application protocol parsers. In: Internet Measurement Conference, Rio de Janeiro, Brazil (2006)
24. Borisov, N., Brumley, D., Wang, H.J., Dunagan, J., Joshi, P., Guo, C.: A generic application-level protocol analyzer and its language. In: Network and Distributed System Security Symposium, San Diego, CA (2007)
25. Caballero, J., Yin, H., Liang, Z., Song, D.: Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In: ACM Conference on Computer and Communications Security, Alexandria, VA (2007)
26. Cui, W., Peinado, M., Chen, K., Wang, H.J., Irun-Briz, L.: Tupni: Automatic reverse engineering of input formats. In: ACM Conference on Computer and Communications Security, Alexandria, VA (2008)
27. Wondracek, G., Comparetti, P.M., Kruegel, C., Kirda, E.: Automatic network protocol analysis. In: Network and Distributed System Security Symposium, San Diego, CA (2008)
28. Lin, Z., Jiang, X., Xu, D., Zhang, X.: Automatic protocol format reverse engineering through context-aware monitored execution. In: Network and Distributed System Security Symposium, San Diego, CA (2008)
29. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Network and Distributed System Security Symposium, San Diego, CA (2005)
30. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D., Engler, D.R.: Exe: Automatically generating inputs of death. In: ACM Conference on Computer and Communications Security, Alexandria, VA (2006)
31. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: SIGPLAN Conference on Programming Language Design and Implementation, Chicago, IL (2005)
32. Godefroid, P., Levin, M.Y., Molnar, D.: Automated whitebox fuzz testing. In: Network and Distributed System Security Symposium, San Diego, CA (2008)
33. : Vine `http://bitblaze.cs.berkeley.edu/vine.html`.
34. Caballero, J., McCamant, S., Barth, A., Song, D.: Extracting models of security-sensitive operations using string-enhanced white-box exploration on binaries. Technical Report UCB/EECS-2009-36, EECS Department, University of California, Berkeley (2009)
35. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: A new approach to computer security via binary analysis. In: International Conference on Information Systems Security, Hyderabad, India (2008) Keynote invited paper.
36. : Towards generating high coverage vulnerability-based signatures with protocol-level constraint-guided exploration (extended version) `http://www.ece.cmu.edu/~juanca/papers/fieldsig_extended.pdf`.
37. Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. In: SIGPLAN Conference on Programming Language Design and Implementation, Tucson, AZ (2008)
38. Boonstoppel, P., Cadar, C., Engler, D.: Rwset: Attacking path explosion in constraint-based test generation. In: International Symposium on Software Testing and Analysis, Seattle, WA (2008)
39. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM **18**(8) (1975)