ShieldGen: Automatic Data Patch Generation for Unknown Vulnerabilities with Informed Probing

Weidong Cui Marcus Peinado Helen J. Wang
Microsoft Research
One Microsoft Way
Redmond, WA 98052
{wdcui, marcuspe, helenw}@microsoft.com

Michael E. Locasto
Columbia University
1214 Amsterdam Avenue
New York, NY 10027
locasto@cs.columbia.edu

Abstract

In this paper, we present ShieldGen, a system for automatically generating a data patch or a vulnerability signature for an unknown vulnerability, given a zero-day attack instance. The key novelty in our work is that we leverage knowledge of the data format to generate new potential attack instances, which we call probes, and use a zero-day detector as an oracle to determine if an instance can still exploit the vulnerability; the feedback of the oracle guides our search for the vulnerability signature. We have implemented a ShieldGen prototype and experimented with three known vulnerabilities. The generated signatures have no false positives and a low rate of false negatives due to imperfect data format specifications and the sampling technique used in our probe generation. Overall, they are significantly more precise than the signatures generated by existing schemes. We have also conducted a detailed study of 25 vulnerabilities for which Microsoft has issued security bulletins between 2003 and 2006. We estimate that ShieldGen can produce high quality signatures for a large portion of those vulnerabilities and that the signatures are superior to the signatures generated by existing schemes.

1. Introduction

Recently, we have seen a rise in zero-day attacks that exploit unknown vulnerabilities [25]. Unfortunately, current practice in new vulnerability analysis and protection generation is mostly manual. In this paper, we aim to automate this process and enable fast, patch-level protection generation for an unknown vulnerability, given the observation of a zero-day exploit of the vulnerability.

In particular, we consider a fast, patch-level protection to be in the form of a *data patch* rather than the more traditional software patch. A data patch serves as a policy for a data filter, and is based on the vulnerability or the software flaw that needs to be protected. The filter uses the data patch to identify parts of the input data to cleanse as it is being consumed. As a result, the sanitized data stream will not exploit the vulnerability. Shield vulnerability signatures [29], which are used by firewalls to filter malicious network traffic, are examples of data patches for network input. Similarly, files can be crafted maliciously to exploit a vulnerability in an application that consumes file input (e.g., WMF vulnerability [26]). With the rise of such exploits, anti-virus software vendors have started using vulnerability signatures for data files to defend against new attack variants. “Data patch” and “vulnerability signature” are used interchangeably in this paper. We use the term “data patch” when emphasizing its purpose as a patch and the term “vulnerability signature” when emphasizing its form as a signature.

By being data-driven, data patches can take the form of signatures that can be automatically distributed and enacted on vulnerable hosts. This style of protection cannot be achieved by the traditional software patches since applying a software patch is inherently user-driven. Even with automatic patch download, users often still need to enact the downloaded patch by restarting the application or rebooting the machine. Furthermore, in an enterprise environment, patches are typically tested prior to deployment in order to avoid the potentially high cost of recovering from a faulty patch. In contrast, rolling back a data patch is as simple as removing the vulnerability signature.

In this paper, we present *ShieldGen*, a system for

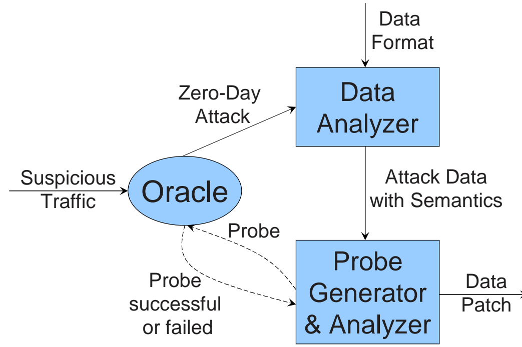

Figure 1. The ShieldGen System Overview

automatically generating a data patch for an unknown vulnerability, given a zero-day attack instance. The key novelty in ShieldGen is that we leverage knowledge of the data format to generate new potential attack instances and use a zero-day detector as an oracle to guide the search of vulnerability signatures. In particular, we assume knowledge of the data format of a zero-day attack, such as the protocol format that is used by a network-based attack or the file format that is used by a file-based attack. This is a reasonable assumption in that the type of data input that is consumed by an application is often known, and it is common practice to have data formats specified for the purpose of interoperability across vendors or simply for the purpose of documentation in a proprietary setting.

In our design, we employ a zero-day attack detector to detect zero-day attacks with high confidence. The detected zero-day attack is sent to our system for producing a data patch. Then, we construct new potential attack instances, which we call *probes*, based on the data format information. We send the probes back to the *oracle*, namely the zero-day attack detector, to see whether these probes succeed as real attacks. Answers from the oracle will then guide our system to construct new probes, to discard attack-specific parts in the original attack data as well as retain the inherent, vulnerability-specific part. The output of our system is a vulnerability signature in the form of a refinement of the data format specification that embeds a vulnerability predicate — a set of boolean conditions on data fields. For stateful network-based attacks, our system is able to capture the protocol context such as the protocol state at which an attack message can be sent. Figure 1 gives an overview of the ShieldGen system.

The number of probes used by ShieldGen for deriving a vulnerability signature is the key measure of its efficiency. Hence, a goal of our probe generation algorithm is to minimize the number of probes. To this end, we leverage semantic information and constraints in the data format specification. For example, we enforce the dependency constraints across data fields so that we will not generate invalid probes.

We have implemented a ShieldGen prototype. For evaluation, we have experimented with the vulnerabilities behind Slammer [17], Blaster [28], and a WMF [26] attack. The generated signatures have no false positives. They have a low rate of false negatives, due to the imprecision of data format specifications and the sampling technique used in our probe generation. We also conducted a pencil-and-paper evaluation on the vulnerability coverage of our approach. We examined 377 vulnerabilities from Microsoft Security Bulletins issued between 2003 and 2006. At least 157 out of these 377 vulnerabilities had data input as an attack vector and are potentially data patchable. We selected 25 vulnerabilities that we can understand from this set and analyzed them in detail. We estimate ShieldGen to be effective for 19 of the 25 vulnerabilities. We also estimate that Vigilante's signature scheme [4] would be effective for only 6 of the 25 vulnerabilities.

The rest of the paper is organized as follows. We first present the related work in Section 2 and position ShieldGen in relation to existing work in automatic signature generation. Then, we give background on the building blocks of our system in Section 3. In Section 4, we present the details of our design. In Section 5, we present our implementation and evaluation that consists of case studies of three vulnerabilities with which we experimented using ShieldGen and a vulnerability coverage study. We discuss future work in Section 6, and finally conclude in Section 7.

2. Related Work

Automatic signature generation for zero-day attacks has received much attention in the research literature. In this section, we compare and contrast ShieldGen with related work in automatic signature generation.

Early efforts such as Autograph [9], Earlybird [24], and Honeycomb [10] are designed to generate attack signatures for a single attack variant, searching for long invariant substrings from network traffic as signatures.

For capturing polymorphic attack variants, Polygraph [20] exemplifies the approach of finding multiple invariant substrings from the network traffic. Polygraph observes that multiple invariant substrings must often be present in all variants of a worm payload for the worm to function properly. These substrings typically correspond to protocol framing, control data like return addresses, and poorly obfuscated code. Polygraph still suffers from significant false positives and false negatives because legitimate traffic often contains multiple invariant substrings, and polymorphic attacks could hijack control data without using an invariant substring [6].

The above work generates signatures from network traffic alone. A fundamental drawback of such mechanisms is that carefully crafted attack traffic can mislead them to generate incorrect signatures [23].

Some existing work leverages information about vulnerable applications for improving both the accuracy and the coverage of signatures. Nemean [32] employs protocol specifications to provide more protocol context to their attack signatures and tries to generalize the signature for observed attack instances. Nemean's signature is a finite state automaton (FSA) inferred from clusters of similar connections or sessions. The edges in a connection-level FSA can be either fields or messages, and the edges in a session-level FSA are connections. It generalizes the signatures by replacing certain variable data elements with a wild card. Unlike ShieldGen, (1) Nemean's generalization is dependent on attack instances observed. This can make the resulting signatures too specific. For example, attack variants that make use of a different message sequence cannot be captured by Nemean. Also, the wild card-based generalization cannot filter attack variants of buffer overrun vulnerabilities. (2) The validity of Nemean's generalization is unchecked, which can lead to over-generalization and false positives. In contrast, ShieldGen generates new attack instances based on a single attack instance along with a data format specification without relying on just observed attack instances, and ShieldGen's signature generalization process is validated by the oracle.

COVERS [11] uses an address-space randomization-based zero-day detector and a regular-expression-based protocol specification to generate signatures for buffer overrun vulnerabilities. A key difference between ShieldGen's signatures and that of COVERS is that ShieldGen's signatures incorporate protocol context in which attacks can happen, such as at what protocol state an attack can happen; COVERS' signatures do not contain any protocol context but only a pattern matching predicate for a particular protocol message. Signatures without protocol context can result in false negatives when different message sequences lead to the same attack and result in false positives when pattern-matching a message at a non-vulnerable protocol state [29]. In COVERS' signature generation, it uses the length of the vulnerable input field as the buffer limit for the buffer overrun condition in its signature; this can cause false negatives for attacks that have shorter buffer length than that of the observed attack instance. In ShieldGen, we find out the buffer limit with the help of our zero-day detector which is based on dynamic data flow analysis. Our experimental results indicate that this buffer limit yields zero false positives and harmless false negatives (Section 4.3.2 and Section 5.1.1). Furthermore, the coverage of ShieldGen's signatures is much greater because of our use of data format-informed probing to the oracle.

Previous research efforts we have discussed so far are dependent on attack instances observed. The Packet Vaccine system [30] breaks this limitation by manipulating packet payloads and observing program reactions to them.

Packet Vaccine generates signatures in three steps: (1) It constructs packet vaccines or probes by randomizing address-like strings. (2) It detects exploit by observing memory exception upon packet vaccine injection. (3) It generates signatures by finding in the attack input the bytes that cannot take random values. In step 3, it constructs packet vaccines for each byte other than the address string by randomizing its value. The similarity between Packet Vaccine and ShieldGen is that they both leverage a feedback loop to improve the coverage of signatures. (Our probing and feedback mechanism was developed independently.) However, compared to ShieldGen, Packet Vaccine has two limitations: (1) Its main probing scheme randomizes each byte rather than leveraging data format information. Compared to ShieldGen's data format-informed probing, its probing strategy suffers from significantly more probes particularly when multiple messages are involved in an attack. In fact, the authors of Packet Vaccine mentioned that their scheme works more reliably for text-based proto-

cols than the binary ones because of the lack of protocol knowledge for binary data formats. The authors of Packet Vaccine briefly mentioned the benefit of leveraging protocol specifications. However, it is unclear what type of protocol specification language is considered and how protocol specifications are leveraged. The latter includes many intricate issues to which most of this paper is devoted. (2) Packet Vaccine can only detect control-flow hijacking attacks while ShieldGen uses a zero-day detector that can detect a wider range of attacks. For example, Packet Vaccine cannot detect exploits of the WMF vulnerability [26].

Newsome and Song [21] hinted the research direction of probing a zero-day detector oracle to find attack invariants. They proposed flipping bits of the original attack data to generate probes. Such a probing method would be prohibitively expensive. ShieldGen's leverage of the data format information scales down the number of probes significantly and is critical for the practicality of such a scheme. Much of this paper is devoted to the techniques on how to leverage the data format information to generate useful probes.

Another category of automatic signature generation work uses program analysis for binary or source code.

Costa *et al.* [4], Crandall *et al.* [6], and Newsome *et al.* [19] use dynamic data flow analysis over the execution on an attack input and generate a signature in the form of symbolic predicates. Such attack signatures are inherently specific to the attack input used in the data flow analysis. In ShieldGen, we generalize attack-specific symbolic predicate-based signatures to cover significantly more attack variants with data format-informed probing to the oracle.

Brumley *et al.* [3] use static program analysis to extract the program logic that processes the attack data and triggers the vulnerability. The extracted logic can be expressed in the form of Turing Machines, symbolic predicates, or regular expressions as vulnerability signatures. Turing Machine-based signatures may not terminate, and regular expressions are not sufficiently expressive for many vulnerabilities. Symbolic predicates are the most practical form. The authors introduce the notion of Monomorphic Execution Path (MEP) and Polymorphic Execution Path (PEP) to describe the coverage of vulnerability signatures. MEP considers the single execution path from the point at which attack input is consumed to the point of compromise while PEP considers many different paths. In fact, the signatures generated by the three systems in the previous paragraph [4, 6, 19] are MEP signatures. We call them *execution trace-based methods* for the rest of the paper.

It has remained an open challenge to generate PEP (the more vulnerability-specific kind of) signatures in the form of symbolic predicates. One challenge is the combinatorial explosion in the number of execution paths. Another challenge is that with potentially large attack data (e.g., a very long file maliciously crafted with many iterative elements), the resulting symbolic predicate will contain a large number of conditions (i.e., the number of conditions grows with the number of iterative elements in the input) many of which are unnecessary and overly restrictive causing high rates of false negatives. Furthermore, in a stateful network-based attack that takes place over a sequence of messages, the vulnerability may only be triggered by the last message, and there may be other message sequences that lead to the same last message. In such a scenario, the symbolic predicate generated over the message sequence will not detect attacks that use other message sequences to reach the same vulnerability. By comparison, the ShieldGen approach can cope with these challenges much more easily with knowledge of the data format.

3. Background

This section gives a brief summary of the two building blocks we use to construct ShieldGen: an oracle and a data analyzer.

3.1. The Oracle: A Zero-day Attack Detector

A zero-day attack detector takes suspicious data as input, and outputs with high confidence whether the data contain an exploit. The suspicious data can be obtained from crash dumps or from a honeyfarm. A zero-day attack detector can take many forms [4, 7, 10, 24]. Detectors based on dynamic data flow analysis [4, 5, 21] instrument the software that is to be monitored and track how its input data (network packets or files) propagate in its address space as the program executes. Detectors of this type can use this information to test for a wide range of vulnerability conditions. For example, a simple condition would be to test before executing a `ret` instruction whether input data have propagated into the return address on the stack. A small set of simple conditions of this type is sufficient to give this type of detectors very broad coverage for low-level control and data flow vulnerabilities. This includes buffer overflows, arbitrary vulnerabilities that result in code injection or overwriting of function pointers or return-to-libc style attacks. On the other hand, data flow detectors are unlikely to

detect higher level vulnerabilities, such as incorrect access control settings, incorrect security user interfaces or sandboxing problems in scripting engines.

In ShieldGen, we use the Vigilante's zero-day detector that is based on dynamic data flow analysis [4]. This detector implements three vulnerability conditions. (1) Tests for *arbitrary execution control (AEC):* The detector tests whether input data is about to be moved into the instruction pointer. This will detect attempts to overwrite return addresses on the stack or function pointers on the stack or heap. (2) Tests for *arbitrary code execution (ACE):* Before executing an instruction, the detector tests whether the instruction depends on the program's input. This detects attempts to execute injected code. (3) Tests for *arbitrary function arguments (AFA):* Before performing certain critical system calls (e.g., creating a process), the detector checks whether certain critical arguments depend on the program's input. In practice, this detector has a low rate of false positives. False positives can be eliminated completely at the expense of higher rates of false negatives by means of a verification procedure [4].

In addition to issuing an alert, the detector can also provide detailed information about the exploit and the vulnerability. This includes the complete data flow history and application state at the moment the vulnerability was detected. The detector can output the positions within the input stream of the values that triggered the alert. For example, in the case of an AEC condition, these are the bytes that were about to be loaded into the instruction pointer. In the case of an ACE condition, these are the bytes that were about to be executed. Furthermore, the detector can output information about the instruction that triggered the alert. For example, the detector will output whether an AEC alert was triggered while executing a `ret` instruction (overwritten return address) or an indirect `jmp` or `call` instruction (overwritten function pointer). We use this information in ShieldGen.

3.2. Data Format Specification and Data Analyzer

We assume knowledge of the data format of the input to the vulnerable application. We also assume the input is not encrypted or obfuscated. We consider two types of data: network data and file data.

A data analyzer is a tool that parses data according to a data format specification, giving semantics and structure to the raw data. A data analyzer can operate either online or offline. An online data analyzer can serve as the main mechanism in data filters that prevent network or file-based intrusions. For example, Shield [29] employs a protocol analyzer to perform vulnerability-driven filtering of application level protocol traffic; a file analyzer can by employed by anti-virus software to prevent file-based attacks that exploit file parsing vulnerabilities.

Recent work, such as binpac [22] and GAPA [2], has advocated specifying the protocol format using a domain-specific language, and then using a *generic* protocol analyzer runtime to parse protocol traffic according to the specification. BinPac and GAPA are able to map the protocol structure over the raw network data, parsing packets into messages that contain various fields as expressed in the specification, and extracting the protocol state information based on the sequence of messages already parsed. In our work, we employ GAPA. We find that GAPA can be readily used for file format specification and analysis. A GAPA specification, in short a *Spec*, specifies message format, protocol state machine, and message handlers (of which there is one per protocol state) and carries out the protocol state transition. The message format is expressed in an enhanced BNF format similar to the one used in RFC's. The run-time value of earlier parts of the message may determine how later parts of the message are parsed. For example, a size field before an item array indicates the number of items to parse at run-time; a field value in one message may determine how to parse parts of a subsequent message. Such context-sensitive characteristics of the data formats are well supported in GAPA through enhancing the BNF notation and embedding code in the BNF rules.

4. Design

4.1. Goals

We have the following goals for ShieldGen's data patch generation:

- *No false positives.* ShieldGen signatures should have no more false positives than the oracle.
- *Minimize the number of false negatives.* The rate of false negatives should be as low as possible.
- *Minimize the number of probes.* ShieldGen should be reasonably efficient.

Ideally, we would also like the signature to be free of false negatives. However, computing a signature that has neither false positives nor false negatives is equivalent to solving the halting problem [3].

```
protocol SQL {
  transport = (1434/UDP);
  grammar {
    SQLMessage -> type:byte rest:".*";
  };

  state-machine SQLClientOrServer {
    (S_Init, IN)->H_SQLMessage;
    (S_Init, OUT)->H_SQLMessage;
    initial_state = S_Init;
    final_state = S_Final;
  };

  handler H_SQLMessage(SQLMessage) {
    /* the if condition below is the
       vulnerability predicate being
       added by ShieldGen */
    @SQLMessage {
       if (type==0x04 && size(rest)>=97)
          return EXPLOIT;
    }
    return S_Final;
  };
};
```

Figure 2. Data Patch for the Vulnerability behind Slammer

4.2. ShieldGen's Data Patch

The data patch generated by ShieldGen is a refinement of the data format specification that is to be fed to a data analyzer of a firewall or an anti-virus program for patch-equivalent protection. The refinement includes a vulnerability predicate on fields of the input. If the predicate evaluates to `true` the input is classified as an exploit and removed. In the GAPA language, the predicate is expressed as a condition in an *if*-statement that is inserted into the appropriate data handler. For network data, the data handler is the message handler at the protocol state at which attacks can happen. To adapt GAPA for file data, the entire file is treated as one message, the protocol state machine specification contains just one state, and there is just one data handler. Figure 2 shows an example data patch for the vulnerability behind the Slammer attack [17].

4.3. Data Patch Generation

4.3.1. Overview

Central to ShieldGen's data patch generation is the derivation of the vulnerability predicate. Figure 3 gives an overview of the data patch generation procedure of ShieldGen.

Our first step is to use our data analyzer and check whether the input (i.e., the attack instance) violates any data format constraints (e.g., the number of bytes in a byte array must correspond to the value of its size field). For the violated constraints, we construct probes that satisfy the constraint (e.g., changing the size field to correspond to the size of the byte array in the original attack packet). If the probe is reported unsuccessful by the oracle, namely it failed to exploit the vulnerability, the data patch will simply be the data format specification that enforces the corresponding data format constraint. If the probe is successful, we move on to the next step.

Our next step is to generate the attack predicate. This predicate is a conjunction of boolean conditions with each data field (of the message used in the attack) equaling the value in the attack input. We can easily obtain the attack predicate by sending the attack input and its data format to the data analyzer.

Of course, this predicate incurs no false positives in attack detection, but is very restrictive and admits only this attack input at its protocol state. The subsequent steps of our algorithm relax or remove conditions that are specific to the original attack input, so that we can admit more attack variants. We do so by generating probes, potential attack variants, based on the data format, and sending them to the oracle. If the oracle determines that a probe is a new attack variant, we adjust the vulnerability predicate so that the confirmed new attack instance can be admitted as well. For example, if all values of a data field have been tried, and the oracle classifies all the corresponding probes as attacks, then that field is a *don't-care* field, and we can remove the condition on the data field from the predicate.

One challenge in probe generation is that we must generate legitimate messages that satisfy protocol semantics including session semantics and cross-field correlation within a message. For example, the session ID field of all message of a session should be the same. Some examples of field correlations within a message are: length field = sizeof(all fields), checksum field = checksum (all fields), hash field = hash (another field). It is important that such semantics are obeyed in our probes; otherwise, the oracle's answer to illegitimate probes could mislead our judgment on the field under experiment. We include as much protocol semantics as we know about a protocol into its Spec. In our prototype, we have implemented protocol semantics for lengths and

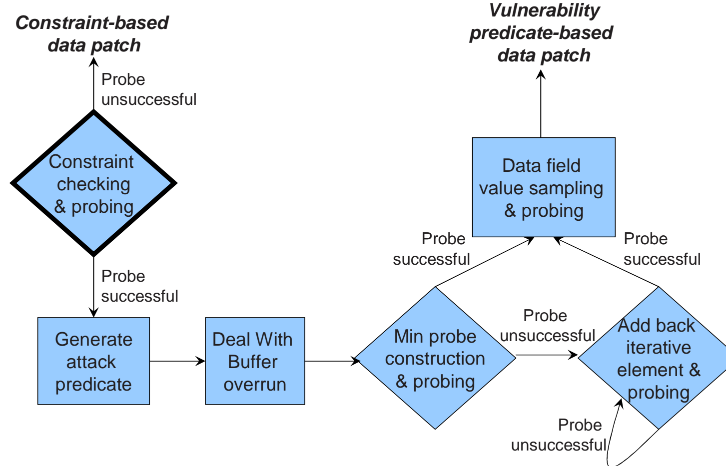

Figure 3. The procedure of ShieldGen's data patch generation: The procedure starts with "Constraint check & probing".

session IDs; previous work RolePlayer [8] has shown that probes that obey these two semantics work well for a wide range of protocols such as FTP, RPC, and SMB. Note that a probe may contain multiple protocol messages including those messages that precede the offending message for session initiation and application context setup.

The efficiency metric of such a probing scheme is the number of probes. Some probes can be parallelized while some need to be sequential, namely, if the construction of a later probe is dependent on the outcome of an earlier probe. Given enough machines, parallel probes can be sped up by evaluating probes in parallel.

It is typically infeasible to probe the oracle with all combinations of values of each data field that appears in the input. Therefore, we reduce the number of probes in the following ways.

First, there may be iterative elements in the attack data, such as a sequence of records which may be expressed as "records → record records | nil;" or "itemArray → size:uint8 items:int[byte]" in the data format Spec. Such iterative elements can also introduce numerous, repetitive data fields which may not all matter to the vulnerability predicate construction (e.g., it could be the case that just a particular iterative element triggers the vulnerability). Because we can recognize iterative elements based on the Spec, we can issue a probe that contains just one element to see whether the number of elements matters; and then based on the oracle's answer, we issue subsequent probes. Being able to recognize iterative elements in the zero-day attack can significantly reduces the number of probes.

Second, by obeying protocol semantics and eliminating illegitimate probes, we can have a reduction in the number of probes needed. It is possible that an expressed constraint may not actually be a constraint in an implementation. For example, an application may allow the size constraint to be violated. We use a probe to determine whether such a constraint matters.

At last, for network input that contains a sequence of messages, it is highly likely that the vulnerability predicate is only dependent on the last message. This is because vulnerabilities are often localized. Given that message handlers for each message are typically separate code pieces, if an attack happens during the handling of a message, the vulnerability likely affects the message handling for that message only. In this case, having conditions on data fields of earlier messages will cause false negatives in the resulting vulnerability predicate which will not capture attack variants that take a different message sequence. Here, we make the assumption that only the content of last message determines whether the input exploits the vulnerability and that the data analyzer naturally traces the protocol context before reaching the handler of the last message. We will discuss exceptions that violate this assumption in Section 6.

4.3.2. Probe Generation Algorithm

Now, we present our probe generation algorithm in detail. When the oracle detects an attack, it also outputs

the byte offset of the offending byte in the input (Section 3.1). In general, the offending byte may fall into an arbitrary field in the input packet.

Buffer overrun heuristic for character strings: We first want to find out whether we are dealing with a buffer overrun vulnerability. We use a heuristic: If the offending byte lies in the middle of a byte or unicode string then ShieldGen diagnoses a buffer overrun and adds the following condition as a refinement.

```
sizeof(buffer) >
offendingByte_offset - bufferStart_offset
```

The argument for why this heuristic is accurate depends on the type of alert the Vigilante detector issued.

Case 1: ACE alerts: If the application was about to execute a `ret` instruction when the alert was issued then a return address on the stack was corrupted by input data. It is almost certain that the cause of this condition was a stack buffer overrun.

If the application was about to execute an indirect `jmp` or `call` instruction then the corresponding function pointer contained input data. Even if this did represent legitimate application behavior (i.e., the application intended to use input data as a function pointer), an accurate Spec would declare this function pointer as a four-byte integer, and not as a substring of a byte or unicode string.

Case 2: AEC alerts: The application was about to execute an instruction that originated in the middle of a byte or unicode string in the input. It appears highly likely that this was caused by a buffer overrun.

Case 3: AFA alerts: The application was about to make a critical system call with a parameter that originated in the middle of a byte or unicode string in the input. If this was legitimate application behavior, this parameter should appear as a separate data field in an accurate Spec. In this case, the parameter would not lie in the middle of a larger string field. Again, this is a strong indication of a buffer overrun.

Iteration removal: Many popular input formats include arbitrary sequences of largely independent elements (*records*). For example, HTML files contain sequences of tags. Most audio and video files consist of sequences of chunks of compressed audio or video data. WMF files consist of a sequence of drawing command records. In all cases, the file contains a sequence of records. Rendering applications typically have different handler functions for different record types. For example, WMF record types include `Ellipse`, `Rectangle` or `Escape`. Each record type is processed by a corresponding handler function. Vulnerabilities are typically located in a particular handler function. In this case, any input that contains a malicious record is an attack — irrespective of what other records may exist in the input. This poses special challenges to execution-trace-based signature generation mechanisms (Section 2).

In ShieldGen, we can cope with such iterative elements much more easily because they are well specified in the Spec. A critical step in our algorithm is to remove all iterative elements that are not necessary to exploit the vulnerability. This not only reduces the number of the probes needed but also improves the accuracy of the generated signature. At a high level, we generate probes from which we have removed some of the iterative elements and feed the probes into the oracle. If the probe still exploits the vulnerability then the iterative elements that were removed in the probe can be omitted. Otherwise, one or more of the iterative elements that had been removed from the probe are necessary and need to be added. This search procedure is described in more detail below.

In the theoretical worst case, this procedure may not lead to any simplification of the input — namely if all iterative elements in the input are necessary to exploit the vulnerability. However, for the real world vulnerabilities we have observed, the procedure rapidly converges to a probe in which only one or two iterative elements are left. This is consistent with our observation that records are independent and that vulnerabilities are typically located in the handler for a particular record type.

In more detail, our search is aided by the fact that the Vigilante detector identifies the offending data in the input. The data analyzer can map this information to a particular iterative element — the *offending element*. Given our heuristic observation that typically only one or two iterative elements are necessary to exploit the vulnerability, we generate the first probe by removing all iterative elements except for the offending element. If the minimal probe does not succeed as an attack, we have to add iterative elements back into the probe to make it work. Iteration can be hierarchical, namely, an iterative element can have another iterative structure in it, and so on. In our algorithm, we first add back all iterative elements from the original input at the lowest hierarchical level where the offending byte is located. We send such a probe to the oracle. If the oracle detects the probe as an attack, we then try to eliminate the iterative elements at the lowest level using a divide-and-conquer mechanism: We split all the iterative elements at this level into

N parts (for some $N > 1$). We construct N parallel probes with each probe dropping one of the N parts. If a probe is successful, the associated iterative elements can be dropped; otherwise, we further divide the $1/N$ portion of the interactive elements N ways and send another round of parallel probes. We do so until we find the iterative elements that must exist to make a probe work.

If the offending byte does not belong to any iterative element, or we still do not have a successful attack after adding iterative elements at all levels around the offending byte, then we need to add the iterative elements that do not contain the offending byte. In this case, we take an approach similar to how we add back elements containing the offending byte. The only difference is that we do it top-down, starting from the highest hierarchical level. We first use the divide-and-conquer mechanism to find what elements at the highest level must be added back. Then for each element that must be added back, we go down one level inside it and use the same mechanism to find out what elements in this lower level must be added back. We continue this procedure until we reach the lowest level.

Our iterative element removal shares the same intuition as the hierarchical delta debugging technique [16], which leverages the hierarchical structure of the input data to expedite the procedure of minimizing failure-inducing inputs for more effective debugging.

Eliminating irrelevant field conditions: Next, we construct probes over the remaining data fields to eliminate *don't-care* fields and to find additional values of the data fields for which the attack succeeds. We cannot afford sending probes in a combinatoric fashion which would result in an exponential number of probes. Instead, we evaluate one field at a time, setting the values of all other fields to their respective values in the original input. We obey the data format semantics in our probe construction as mentioned earlier.

We use the following sampling heuristics to further reduce the number of probes for evaluating each field: (1) For any base type data field such as integer, we try its minimum value, its maximum value, and sample some number of random values below and above the input value. (2) For character strings, we generate a random string by taking a random non-zero value for each byte; and we send several samples of such random strings. (3) For byte arrays, we use a random value for each byte rather than non-zero value.

When all samples for a field lead to probes that exploit the vulnerability, we consider the field as a *don't-care* and remove it from the predicate. Our sampling can produce both false negatives and false positives. In our evaluation, we have not observed any cases in which sampling introduces false positives. As discussed in Section 6, it may be possible to avoid this problem by leveraging Vigilante filter conditions.

At the end of this process we are left with a set of input fields f_i and values for each field V_i, such that the attack will be triggered if value$(f_i) \in V_i$ for all i. The output of our algorithm is the conjunction of these conditions. This is our approximation of the vulnerability predicate. It is possible to have more complex vulnerability conditions that involve complex calculations on multiple fields. We discuss possible approaches to such complex conditions in Section 6.

5. Implementation and Evaluation

We have developed a prototype of ShieldGen. Our prototype system consists of a re-implementation of the Vigilante [4] zero-day detector as our oracle, the GAPA [2] data analyzer, and a 1,500 line perl script that implements probe generation and vulnerability predicate derivation.

We evaluate ShieldGen on its accuracy, efficiency, and applicability in two ways. First, we conduct three case studies, using ShieldGen to generate data patches for three known vulnerabilities. We evaluate the accuracy of the data patches and efficiency of their generation. Second, we conduct a pencil-and-paper vulnerability study to estimate ShieldGen's coverage as well as its potential accuracy. We measure efficiency in terms of the number of parallel probes and the number of sequential probes (Section 4.3). Each probe takes about 10 seconds; we believe there is much room for efficiency improvement for probe processing as we discuss in more detail in Section 6.

5.1. Case Studies

We ran ShieldGen for three well known vulnerabilities: the vulnerabilities behind Slammer [17] in the SQL Server 2000 Resolution Service and Blaster [28] in the Windows RPC service, and a Windows Metafile (WMF) vulnerability [26]. We chose these case studies because they cover both file-based and network-based vulnerabilities. For the latter, they cover both single message and multiple-message based vulnerabilities: The vulnerable WMF application uses file input while the vulnerable SQL and RPC services take network input; exploits against the SQL vulnerability are very simple and re-

quire just one packet while exploits against the RPC vulnerability are more complex and require a sequence of messages.

We conducted our case study in an isolated testbed of three virtual machines (VMs) running Microsoft Virtual PC [15]. One VM is installed with an unpatched Windows 2000 Server and runs the Vigilante-instrumented SQL service and RPC service, serving as the oracle for attacks exploiting the vulnerabilities behind Slammer and Blaster. We use the second VM to send network probes to the SQL or RPC service. The third VM plays the role of the oracle for the attacks against the WMF vulnerability. The VM is installed with an unpatched version of Windows XP Professional (Service Pack 2). We use the Windows Picture and Fax Viewer as the WMF application. The WMF probes are generated in the same VM.

In our current prototype, during the base type field sampling phase of the probe generation (Section 4.3), we sample the minimum value, maximum value, one random value that is smaller than the corresponding value in the original attack data, and one random value that is larger, if applicable. For strings and byte arrays, we sample a random string or byte array for three times.

5.1.1. The SQL Vulnerability

The SQL vulnerability [12] is a stack buffer overrun vulnerability in the SQL Server 2000 Resolution Service (SSRS), which provides server resolution service for multiple SQL server instances running on the same machine. SSRS listens for requests on port 1434/udp and returns the IP address and port number of the SQL server instance that runs the requested database.

The message format of the UDP request consists of just two fields, a byte followed by a byte array. We obtained the original attack trace by launching an attack from a standalone program that replaces the original self-propagation payload in the Slammer worm with a windows shell.

ShieldGen determined that the attack is a stack buffer overrun from the output of Vigilante and constructed a buffer overrun vulnerability condition. Then, ShieldGen further issued four parallel probes to try different values on the first byte. ShieldGen successfully generated the correct vulnerability signature as shown in Figure 2. This signature adds two refinements to the GAPA Spec: the first byte must be 0x04 and the minimal size of the byte array must be 97 bytes. This signature yields no false positives but may have *harmless* false negatives. These false negatives would have input longer than the allocated buffer size but shorter than the minimal size to overwrite the return address. Thus these false negatives will not be able to compromise the system. The Vigilante signature is similarly accurate.

5.1.2. The RPC Vulnerability

The RPC vulnerability [13] is a stack buffer overrun vulnerability in the RemoteActivation and the ISystemActivator interface in Microsoft's DCOM RPC service. We experimented only with the RemoteActivation interface.

We obtained the RPC GAPA Spec, including the format for the RemoteActivation interface. We used the Metasploit framework [18] to generate an attack instance.

ShieldGen determined that this is a stack buffer overrun, and constructed a buffer overrun vulnerability condition based on the Vigilante output: The minimal size of the buffer field for a successful attack must be 40 bytes. We probed that the size and session ID constraints do matter in the RPC implementation. Our probes satisfy these constraints. ShieldGen issued 107 parallel probes to find if other fields than the buffer field are *don't-care* fields. The vulnerability condition excluding the buffer overrun is the conjunction of the following conditions: `rpc_vers == 0x5`, `rpc_minor == 0x01`, `packet_drep == 0x10`, `version_major == 0x5`, `extension_ptr == 0`, `offset == 0`. Due to lack of space, we show the signature at [1]; the signature has about 300 lines.

Our signature has no false positives, but has false negatives due to the conditions `extension_ptr == 0` and `offset == 0`. These false negatives arose because our GAPA Spec was not complete. While sampling values for `extension_ptr`, our probes did not include an extension record in the message. In the case of `offset`, we did not capture the offset constraint. The other conditions are benign based on our understanding of the RPC protocol.

For comparison, the filter produced by the Vigilante filter generator contains conditions on 22 additional fields that ShieldGen has classified as *don't-cares*.

5.1.3. The WMF Vulnerability

A Windows Metafile (WMF) file contains a sequence of records that map to Graphic Display Interface (GDI) functions to create images. For example, a Rectangle record in a WMF file tells the graphics rendering library to draw a rectangle in the image. The WMF vulnerability [14] lies in a SETABORTPROC `Escape` record. An attacker can create a malicious WMF file by adding

a SETABORTPROC `Escape` record with a byte array as a field in the record. The content of the byte array is treated as executable code (for the abort procedure). When the file is opened, the code in the byte array will be executed.

We developed a GAPA Spec for the WMF file format. WMF records are iterative elements in the Spec. We wrote a standalone program that creates malicious WMF files by adding a SETABORTPROC Escape record (with a piece of code) in the middle of a randomly generated sequence of regular draw records.

ShieldGen first determined that this is not a buffer overrun attack because the offending byte is the first byte of a byte array that stores the piece of code. ShieldGen then issued a probe after removing all iterative elements and found that it failed. Next, ShieldGen issued $log(M)$ (M is the number of regular draw records in the attack file) sequential probes to add back iterative elements. It found that there must be one regular draw record following the Escape record; otherwise, the application does not execute the code. It then issued 72 parallel probes to find if any field left in the minimal attack file is a *don't-care*. In addition to the requirement of a regular draw record following the Escape record, the vulnerability predicate contains a conjunction of `mf_type == 0x1`, `version_number == 0x0300`, `escape_num == 0x9`, and the file must be ended with an EOF record. Due to lack of space, we show the signature at [31]; the signature has about 130 lines.

A close inspection of the vulnerability at the source code level reveals that these conditions are too strong. There is one other value for both the `mf_type` and `version_number` field for which an attack would succeed. ShieldGen did not find these values because it only samples field values, rather than exhaustively searching them.

The filter produced by the original Vigilante filter generator contains conditions for each record in the WMF file — including benign draw records. The filter is, thus, attack specific. In contrast, ShieldGen correctly removed all the unnecessary iterative data fields.

5.2. Vulnerability Coverage

Methodology: We have performed a pencil-and-paper vulnerability study in order to evaluate ShieldGen on a larger sample of real-world vulnerabilities. More precisely, we are trying to determine whether the following claims can be made for a significant number of real-world vulnerabilities: (1) The overall approach of automatic data patch generation is applicable. (2) Iteration removal is relevant and ShieldGen can successfully remove iteration from the input. (3) The filter condition produced by ShieldGen is accurate. Furthermore, we will compare the filter quality of ShieldGen with that of execution trace methods.

To cover a larger sample of vulnerabilities, we had to limit the amount of effort needed per vulnerability. Thus, we did not run ShieldGen, which would have required building exploits and GAPA Specs for each vulnerability. Even so, the analysis is quite work intensive and requires understanding input formats and low-level application behavior for a very diverse set of applications.

It was not always clear how to count vulnerabilities. In a number of cases, the fault lies in lower level code such as dynamically linked libraries (DLLs) that can be used by any number of applications. We counted such cases as a single vulnerability and evaluated it against the application that seemed most likely to be affected.

Vulnerability classification: As a first step, we examined 377 vulnerabilities for which Microsoft has issued security bulletins between 2003 and 2006. We found 157 vulnerabilities that can be exploited by either network or file input. These vulnerabilities can in principle be shielded with data patches. The remaining vulnerabilities fall into the following categories: (1) denial of service vulnerabilities and crashes (55 vulnerabilities); (2) access control problems (25 vulnerabilities); (3) scripting problems (17 vulnerabilities); (4) miscellaneous problems (34 vulnerabilities). We were not able to classify 89 vulnerabilities because we did not have enough information about them.

Filter quality of ShieldGen: Next, we chose 25 vulnerabilities from the 157 vulnerabilities that can be shielded by data patches. Our choice was primarily based on whether we could analyze a vulnerability with reasonable effort — which depended on the amount of information available and on our knowledge of the applications and input formats involved. We performed a detailed pencil-and-paper analysis of these 25 vulnerabilities and tried to determine if ShieldGen could generate precise signatures for them. Table 1 displays the results of our analysis. We begin by considering iteration removal. Seventeen of the 25 vulnerabilities are associated with complex input formats that admit iteration. Examples include different multimedia file formats (e.g., WMF, WMV), HTML web pages, word processor and spread sheet files. ShieldGen can remove all unnecessary iterative elements in exploits for all 17 vulnera-

	ShieldGen	Execution Trace Methods
Precise Filter	19	6
Imprecise Filter	6	19
Total	25	25

Table 1. Our assessment of ShieldGen's coverage.

bilities. Most of the remaining eight vulnerabilities for which iteration removal appears irrelevant involve RPC calls.

After iteration has been removed, ShieldGen has to compute filter conditions for the remaining fields. Next, we consider the quality of those conditions. We consider ShieldGen sufficiently precise for a vulnerability if the union of a small number of ShieldGen data patches covers the vulnerability completely without suffering from false positives. Ideally, a single data patch will be able to cover the vulnerability. However, several vulnerabilities in our sample were exploitable from a small number of independent interfaces (between 2 and 10) — requiring the same number of data patches. For example, the RPC (Blaster) vulnerability can be exploited by means of two different RPC calls. We believe that, in a practical setting, ShieldGen data patches will be considered precise if they cover the vulnerability after being given an attack instance for each of the two RPC calls. Based on this criterion, ShieldGen produces precise filters for 19 of the 25 vulnerabilities.

Failure analysis: The six vulnerabilities for which ShieldGen does not produce precise filters can be categorized as follows:

- Complex conditions: For several vulnerabilities, the precise condition involves functions of several input fields. For example, in one case the vulnerability is triggered if and only if the combined length of two separate strings in the input exceeds a certain limit. In another case, the vulnerability is triggered if and only if the value of one integer field in the input is larger than the value of another integer field.

- Unchecked array indices: In two cases, the application uses a field from the input as an index into an array without checking whether the index falls within the bounds of the array. Without an externally provided specification, it is notoriously hard to infer what the array bounds are.

- In one case, the application uses a collection of old buffers whose size does not match the requirements of the data it is currently processing. This gives rise to a large number of variants of buffer overflows — depending on the data for which the buffers had been originally allocated. Conditions of this type are hard to even formulate and very hard to infer automatically.

Comparison with execution trace methods: Next, we estimate how existing filter generation algorithms that are based on execution traces [3, 4, 6, 19] would behave on the vulnerabilities in our sample. We assume that these algorithms can detect simple loops. Such a mechanism is described in [3] and could easily be added to other schemes. Without such a mechanism, filters are specific to the lengths of the strings in the attack instance from which they were generated. Such filters would be fragile.

For the 17 vulnerabilities for which iteration removal is relevant, the execution path of the application can be strongly manipulated by the attacker. As outlined earlier, the attacker can generate exploits by adding arbitrary sequences of iterative elements. Each sequence of iterative elements will produce a corresponding sequence of conditions in an execution trace filter. This makes execution trace filters for these 17 vulnerabilities very fragile and exploit specific.

We estimate that execution trace methods could generate precise filters for about six of the remaining eight vulnerabilities. The other two vulnerabilities involve complex conditions that are associated with multiple executions paths.

We did not try to determine what filters the schemes in [11, 30, 32] would produce for each of the 25 vulnerabilities. Based on the general comparison in Section 2, we believe that, overall, those filters are less precise than ShieldGen filters.

6. Discussion and Future Work

Quality of the data format specification: For our scheme, the quality of the data format specification matters. For example, an unspecified constraint can result in wasted probes and yield unneeded vulnerability conditions that cause false negatives. As another example, if a specification carelessly lumps multiple fields together into a byte array, we would not have the right semantic information about the data — ShieldGen may not be able to determine whether a buffer overrun occurred

(Section 4.3). While obtaining a high quality data format specification is not easy, such a specification is often desirable for many different purposes and could be a one-time effort. It is also easy to imagine that such a high quality specification can be obtained from the context of protocol or file format compliance testing.

Complex filter conditions: In our vulnerability coverage study, the most frequent reason why ShieldGen failed to produce a precise filter was the complex nature of one or more of the filter conditions. ShieldGen uses the detector as a black box, yes/no oracle. There are fundamental limits on the complexity of the conditions that can be inferred in this way. In practice, deriving even simple conditions involving more than one field (e.g., `field-A + field-B > 256`) would require more oracle queries than is feasible.

In contrast, execution trace methods use knowledge about the vulnerable program and the execution trace that leads to the vulnerability in computing their filter conditions. Indeed, these methods can infer arbitrarily complex filter conditions — as long as they can be computed along a single execution path. This suggests the possibility of improving ShieldGen data patches by deriving filter conditions from an execution trace filter generator such as [3, 4, 6, 19].

In principle, this is straightforward. Vigilante filters are symbolic predicates on the values at certain offsets in the input. ShieldGen knows how to map such offsets to the fields defined by the Spec — thus obtaining filter conditions for these fields. However, it is not clear whether these conditions are an improvement over the conditions that ShieldGen can generate on its own. The former encode a single execution path, whereas the latter may correspond to arbitrarily many execution paths that trigger the vulnerability. It is possible to identify such cases by refining the search logic in ShieldGen—but we leave the details to future work.

Probing time: In our current implementation, testing a probe involves starting the application, attaching the detector, sending the probe as input to the application, waiting for the result and shutting down the application. The total time per probe is about ten seconds. We estimate that this time could be reduced to about two seconds per probe with the help of flash cloning of virtual machines [27]. The idea is to create a reference VM in which the application and the detector have already been started. The probes would be tested in clones of the reference VM. After the test, the cloned VM would be discarded. The overhead for cloning and discarding a VM appears to be less than one second [27] — significantly less than the startup and shutdown times of our target applications and the detector.

Attacks not delivered by the last message: Currently we assume the vulnerability can only occur in the message handling code of the last message and any message sequence that leads to the same message handling will also trigger the vulnerability. In practice, there exist exceptional cases. For example, an attack on an FTP server may first create a directory with a long name and then cause a buffer overrun in a subsequent directory listing operation. ShieldGen can handle this kind of attack because Vigilante can return the byte offset of the offending byte in the attack data even when it is not in the last message. In this case, we just need to generate probes on messages starting from the one that contains the offending byte.

7. Concluding Remarks

In this paper, we have presented ShieldGen, a system for automatically generating data patches for an unknown vulnerability, given a zero-day attack instance. The key contribution of our paper is to leverage data format information to construct new attack instances, the probes, and use a zero-day attack detector as an oracle to guide our search for the vulnerability signature. To make such a system practical, we used a number of techniques to reduce the number of probes needed including eliminating iterative elements, obeying data format constraints, and leveraging protocol context. We have implemented a working prototype and experimented with three known vulnerabilities and their respective attack instances. We were able to generate high quality vulnerability signatures efficiently. Our signatures do not contain many *don't-care* data fields that are present in signatures generated by existing execution trace methods. Therefore, our scheme could yield signatures with significantly fewer false negatives. The false negatives come from imperfect data format specifications and the sampling technique used in our probe generation. We also conducted a study of 377 vulnerabilities that were published between 2003 and 2006. We estimate ShieldGen to have significant coverage of data-patchable vulnerabilities with superior accuracy when compared with execution trace-based signatures.

Acknowledgments

We would like to thank Manuel Costa and Miguel Castro for many useful discussions. We thank Laurent Visconti for generating Vigilante signatures for us. We thank Jedidiah Crandall and Dawn Song for their valuable comments on a draft of this paper. We would also like to thank our shepherd and the anonymous reviewers for their detailed suggestions and insightful comments.

References

[1] Data patch for the rpc vulnerability. http://research.microsoft.com/research/shield/papers/blasterFilter.htm, November 2006.

[2] N. Borisov, D. J. Brumley, H. J. Wang, J. Dunagan, P. Joshi, and C. Guo. A Generic Application-Level Protocol Analyzer and its Language. In *Proceedings of the 14th Symposium on Network and Distributed System Security (NDSS)*, Feburary 2007.

[3] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards Automatic Generation of Vulnerability-Based Signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, May 2006.

[4] M. Costa, J. Crowcroft, M. Castro, and A. Rowstron. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, October 2005.

[5] J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37)*, December 2004.

[6] J. R. Crandall, Z. Su, S. F. Wu, and F. T. Chong. On Deriving Unknown Vulnerabilities from Zero-Day Polymorphic and Metamorphic Worm Exploits. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, November 2005.

[7] W. Cui, V. Paxson, and N. Weaver. GQ: Realizing a System to Catch Worms in a Quarter Million Places. Technical Report TR-06-004, ICSI, 2006.

[8] W. Cui, V. Paxson, N. C. Weaver, and R. H. Katz. Protocol-Independent Adaptive Replay of Application Dialog. In *Proceedings of the 13th Network and Distributed System Security Symposium*, February 2006.

[9] H.-A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[10] C. Kreibich and J. Crowcroft. Honeycomb — Creating Intrusion Detection Signatures Using Honeypots. In *Proceedings of ACM SIGCOMM HotNets-II Workshop*, November 2003.

[11] Z. Liang and R. Sekar. Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, November 2005.

[12] Microsoft. Microsoft security bulletin ms02-039. http://www.microsoft.com/technet/security/bulletin/MS02-039.mspx.

[13] Microsoft. Microsoft security bulletin ms03-026. http://www.microsoft.com/technet/security/bulletin/MS03-026.mspx.

[14] Microsoft. Microsoft security bulletin ms06-001. http://www.microsoft.com/technet/security/bulletin/MS06-001.mspx.

[15] Microsoft. Microsoft Virtual PC 2004. http://www.microsoft.com/windows/virtualpc/default.mspx.

[16] G. Misherghi and Z. Su. HDD: Hierarchical Delta Debugging. In *Proceedings of the 28th International Conference on Software Engineering*, May 2006.

[17] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer Worm. *IEEE Magazine of Security and Privacy*, August 2003.

[18] H. Moore. Metasploit Framework. http://www.metasploit.com/.

[19] J. Newsome, D. Brumley, and D. Song. Vulnerability–Specific Execution Filtering for Exploit Prevention on Commodity Software. In *Proceedings of the* 13th *Symposium on Network and Distributed System Security (NDSS 2006)*, February 2006.

[20] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, May 2005.

[21] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 12 Network and Distributed System Security Symposium*, February 2005.

[22] R. Pang, V. Paxson, R. Somer, and L. Peterson. binpac: A yacc for Writing Application Protocol Parsers. In *Proceedings of Internet Measurement Conference*, October 2006.

[23] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif. Misleading Worm Signature Generators Using Deliberate Noise Injection. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, May 2006.

[24] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, December 2004.

[25] The SANS Institute. The Top 20 Most Critical Internet Security Vulnerabilities - PRESS UPDATE. http://www.sans.org/top20/2005/spring_2006_update.php, 2006.

[26] US-CERT. Microsoft Windows Metafile handler SETABORTPROC GDI Escape vulnerability. http://www.kb.cert.org/vuls/id/181038.

[27] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft,

A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, October 2005.

[28] W32.Blaster.Worm. http://securityresponse.symantec.com/avcenter/venc/data/w32.blaster.worm.html.

[29] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *Proceedings of the ACM SIGCOMM*, August 2004.

[30] X. Wang, Z. Li, J. Xu, M. K. Reiter, C. Kil, and J. Y. Choi. Packet Vaccine: Black-box Exploit Detection and Signature Generation. In *Proceedings of the ACM Conference on Computer and Communications Security*, November 2006.

[31] Data Patch for the WMF Vulnerability. http://research.microsoft.com/research/shield/papers/wmfFilter.htm, November 2006.

[32] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha. An Architecture for Generating Semantics-Aware Signatures. In *Proceedings of the 14th USENIX Security Symposium*, July 2005.