Panalyst: Privacy-Aware Remote Error Analysis on Commodity Software

Rui Wang[†], *XiaoFeng Wang*[†] *and Zhuowei Li*[‡]
[†]*Indiana University at Bloomington,* [‡]*Center for Software Excellence, Microsoft*
{*wang63,xw7*}*@indiana.edu, zhuowei.li@microsoft.com*

Abstract

Remote error analysis aims at timely detection and remedy of software vulnerabilities through analyzing runtime errors that occur on the client. This objective can only be achieved by offering users effective protection of their private information and minimizing the performance impact of the analysis on their systems without undermining the amount of information the server can access for understanding errors. To this end, we propose in the paper a new technique for privacy-aware remote analysis, called *Panalyst*. Panalyst includes a client component and a server component. Once a runtime exception happens to an application, Panalyst client sends the server an initial error report that includes only public information regarding the error, such as the length of the packet that triggers the exception. Using an input built from the report, Panalyst server performs a taint analysis and symbolic execution on the application, and adjusts the input by querying the client about the information upon which the execution of the application depends. The client agrees to answer only when the reply does not give away too much user information. In this way, an input that reproduces the error can be gradually built on the server under the client's consent. Our experimental study of this technique demonstrates that it exposes a very small amount of user information, introduces negligible overheads to the client and enables the server to effectively analyze an error.

1 Introduction

Remote analysis of program runtime errors enables timely discovery and patching of software bugs, and has therefore become an important means to improve software security and reliability. As an example, Microsoft is reported to fix 29 percent of all Windows XP bugs within Service Pack 1 through its Windows Error Reporting (WER) utility [20]. Remote error analysis is typically achieved by running an error reporting tool on a client system, which gathers data related to an application's runtime exception (such as a crash) and transmits them to a server for diagnosis of the underlying software flaws. This paradigm has been widely adopted by software manufacturers. For example, Microsoft relies on WER to collect data should a crash happen to an application. Similar tools developed by the third party are also extensively used. An example is BugToaster [27], a free crash analysis tool that queries a central database using the attributes extracted from a crash to seek a potential fix. These tools, once complemented by automatic analysis mechanisms [44, 34] on the server side, will also contribute to quick detection and remedy of critical security flaws that can be exploited to launch a large-scale cyber attack such as Worm epidemic [47, 30].

The primary concern of remote error analysis is its privacy impact. An error report may include private user information such as a user's name and the data she submitted to a website [9]. To reduce information leaks, error reporting systems usually only collect a small amount of information related to an error, for example, a snippet of the memory around a corrupted pointer. This treatment, however, does not sufficiently address the privacy concern, as the snippet may still carry confidential data. Moreover, it can also make an error report less informative for the purpose of rapid detection of the causal bugs, some of which could be security critical. To mitigate this problem, prior research proposes to instrument an application to log its runtime operations and submit the sanitized log once an exception happens [25, 36]. Such approaches affect the performance of an application even when it works normally, and require nontrivial changes to the application's code: for example, Scrash [25] needs to do source-code transformation, which makes it unsuitable for debugging commodity software. In addition, these approaches still cannot ensure that sufficient information is gathered for a quick identification of critical security flaws. Alternatively, one can analyze a vulner-

able program directly on the client [29]. This involves intensive debugging operations such as replaying the input that causes a crash and analyzing an executable at the instruction level [29], which could be too intrusive to the user's normal operations to be acceptable for a practical deployment. Another problem is that such an analysis consumes a large amount of computing resources. For example, instruction-level tracing of program execution usually produces an execution trace of hundreds of megabytes [23]. This can hardly be afforded by the client with limited resources, such as Pocket PC or PDA.

We believe that a good remote analyzer should help the user effectively control the information to be used in an error diagnosis, and avoid expensive operations on the client side and modification of an application's source or binary code. On the other hand, it is also expected to offer sufficient information for automatic detection and remedy of critical security flaws. To this end, we propose Panalyst, a new technique for privacy-aware remote analysis of the crashes triggered by network inputs. Panalyst aims at automatically generating a new input on the server side to accurately reproduce a crash that happens on the client, using the information disclosed according to the user's privacy policies. This is achieved through collaboration between its client component and server component. When an application crashes, Panalyst client identifies the packet that triggers the exception and generates an initial error report containing nothing but the public attributes of the packet, such as its length. Taking the report as a "taint" source, Panalyst server performs an instruction-level taint analysis of the vulnerable application. During this process, the server may ask the client questions related to the content of the packet, for example, whether a tainted branching condition is true. The client answers the questions only if the amount of information leaked by its answer is permitted by the privacy policies. The client's answers are used by the server to build a new packet that causes the same exception to the application, and determine the property of the underlying bug, particularly whether it is security critical.

Panalyst client measures the information leaks associated with individual questions using *entropy*. Our privacy policies use this measure to define the maximal amount of information that can be revealed for individual fields of an application-level protocol. This treatment enables the user to effectively control her information during error reporting. Panalyst client does not perform any intensive debugging operations and therefore incurs only negligible overheads. It works on commodity applications without modifying their code. These properties make a practical deployment of our technique plausible. In the meantime, our approach can effectively analyze a vulnerable application and capture the bugs that are exploitable by malicious inputs. Panalyst can be used by software manufacturers to demonstrate their "due diligence" in preserving their customers' privacy, and by a third party to facilitate collaborative diagnosis of vulnerable software.

We sketch the contributions of this paper as follows:

- *Novel framework for remote error analysis.* We propose a new framework for remote error analysis. The framework minimizes the impact of an analysis to the client's performance and resources, lets the user maintain a full control of her information, and in the meantime provides her the convenience to contribute to the analysis the maximal amount of information she is willing to reveal. On the server side, our approach interleaves the construction of an accurate input for triggering an error, which is achieved through interactions with the client, and the analysis of the bug that causes the error. This feature allows our analyzer to make full use of the information provided by the client: even if such information is insufficient for reproducing the error, it helps discover part of input attributes, which can be fed into other debugging mechanisms such as fuzz testing [35] to identify the bug.

- *Automatic control of information leaks.* We present our design of new privacy policies to define the maximal amount of information that can be leaked for individual fields of an application-level protocol. We also developed a new technique to enforce such policies, which automatically evaluates the information leaks caused by responding to a question and then makes decision on whether to submit the answer in accordance with the policies.

- *Implementation and evaluations.* We implemented a prototype system of Panalyst and evaluated it using real applications. Our experimental study shows that Panalyst can accurately restore the causal input of an error without leaking out too much user information. Moreover, our technique has been demonstrated to introduce nothing but negligible overheads to the client.

The rest of the paper is organized as follows. Section 2 formally models the problem of remote error analysis. Section 3 elaborates the design of Panalyst. Section 4 describes the implementation of our prototype system. Section 5 reports an empirical study of our technique using the prototype. Section 6 discusses the limitations of our current design. Section 7 presents the related prior research, and Section 8 concludes the paper and envisions the future research.

2 Problem Description

We formally model the problem of remote error analysis as follows. Let $P : S \times I \rightarrow S$ be a program that maps an initial process state $s \in S$ and an input $i \in I$ to an end state. A state here describes the data in memory, disk and register that are related to the process of P. A subset of S, E_b, contains all possible states the process can end at after an input exploits a bug b.

Once P terminates at an error state, the client runs an error reporting program $G : I \rightarrow R$ to generate a report $r \in R$ for analyzing P on the server. The report must be created under the constraints of the computing resources the client is able or willing to commit. Specifically, $C_t : \{G\} \times I \times R \rightarrow \Re$ measures the delay experienced by the user during report generation, $C_s : \{G\} \times I \times R \rightarrow \Re$ measures the storage overhead, and $C_n : \{G\} \times I \times R \longrightarrow \Re$ measures the bandwidth used for transmitting the report. To produce and submit a report $r \in R$, the computation time, storage consumption and bandwidth usage must be bounded by certain thresholds: formally, $(C_t(G, i, r) \leq Th_t) \wedge (C_s(G, i, r) \leq Th_s) \wedge (C_w(G, i, r) \leq Th_w)$, where Th_t, Th_s and Th_w represent the thresholds for time, storage space and bandwidth respectively. In addition, r is allowed to be submitted only when the amount of information it carries is acceptable to the user. This is enforced using a function $L : R \times I \rightarrow \Re$ that quantifies the information leaked out by r, and a threshold Th_l. Formally, we require $L(r, i) \leq Th_l$.

The server runs an analyzer $D : R \rightarrow I$ to diagnose the vulnerable program P. D constructs a new input using r to exploit the same bug that causes the error on the client. Formally, given $P(i) \in E_b$ and $r = G(i)$, the analyzer identifies another input i' from r such that $P(i') \in E_b$. This is also subject to resource constraints. Specifically, let $C'_t : \{D\} \times R \times I \rightarrow \Re$ be a function that measures the computation time for running D and $C'_s : \{D\} \times R \times I \rightarrow \Re$ that measures the storage overhead. We have: $(C'_t(D, r, i') \leq Th'_t) \wedge (C'_s(D, r, i') \leq Th'_s)$, where Th'_t and Th'_s are the server's thresholds for time and space respectively.

A solution to the above problem is expected to achieve three objectives:

- *Low client overheads.* A practical solution should work effectively under very small Th_t, Th_s and Th_w. Remote error analysis aims at timely detecting critical security flaws, which can only be achieved when most clients are willing to collaborate in most of the time. However, this will not happen unless the client-side operations are extremely lightweight, as clients may have limited resources and their workloads may vary with time. Actually, customers could be very sensitive to the overheads brought in by error reporting systems. For example, advice has been given to turn off WER on Windows Vista and Windows Mobile to improve their performance [12, 17, 13]. Therefore, it is imaginable that many may stop participating in error analysis in response to even a slight increase of overheads. As a result, the chance to catch dangerous bugs can be significantly reduced.
- *Control of information leaks.* The user needs to have a full control of her information during an error analysis. Otherwise, she may choose not to participate. Indispensable to this objective is a well-constructed function L that offers the user a reasonable measure of the information within an error report. In addition, privacy policies built upon L and a well-designed policy enforcer will automate the information control, thereby offering the user a reliable and convenient way to protect her privacy.
- *Usability of error report.* Error reports submitted by the user should contain ample information to allow a new input i' to be generated within a short period of time (small Th'_t) and at a reasonable storage overhead (small Th'_s). The reports produced by the existing systems include little information, for example, a snapshot of the memory around a corrupted pointer. As a result, an analyzer may need to exhaustively explore a vulnerable program's branches to identify the bug that causes the error. This process can be very time-consuming. To improve this situation, it is important to have a report that gives a detailed description about how an exploit happens.

In Section 3, we present an approach that achieves these objectives.

3 Our Approach

In this section, we first present an overview of Panalyst and then elaborate on the designs of its individual components.

3.1 Overview

Panalyst has two components, client and server. Panalyst client logs the packets an application receives, notifies the server of its runtime error, and helps the server analyze the error by responding to its questions as long as the answers are permitted by the user's privacy policies. Panalyst server runs an instruction-level taint analysis on the application's executable using an empty input, and evaluates the execution symbolically [37] in the meantime. Whenever the server encounters a tainted value that affects the choice of execution paths or memory access,

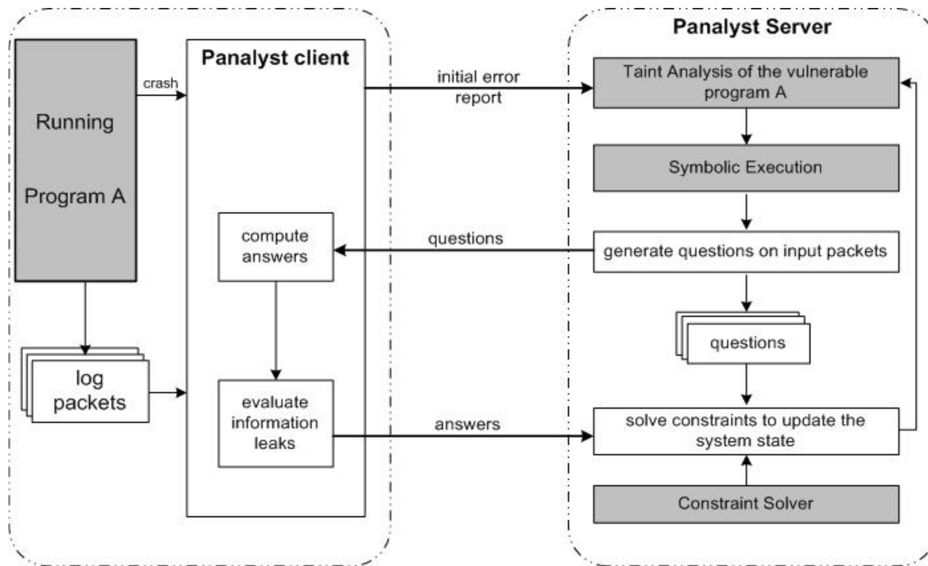

Figure 1: The Design of Panalyst.

it queries the client using the symbolic expression of that value. From the client's answer, the server uses a constraint solver to compute the values of the input bytes that taint the expression. We illustrate the design of our approach in Figure 1.

```
1  If(strcmp(conn->requestMethod, "POST") == 0){
2      buf = malloc(conn->ContentLength+1024);
3      for(len=0;;) {
4          recv(sd, buf+len, 1, 0);
5          len++;
6          if(buf[len-1] == '\n') break;
7      }
8  }
```

Figure 2: An Illustrative Example.

An example. Here we explain how Panalyst works through an example, a program described in Figure 2. The example is a simplified version of Null-HTTPd [8]. It is written in `C` for illustration purpose: Panalyst actually is designed to work on binary executables. The program first checks whether a packet is an HTTP `POST` request. If so, it allocates a buffer with the size computed by adding 1024 to an integer derived from the `Content-Length` field and moves the content of the request to that buffer. A problem here is that a buffer overflow can happen if `Content-Length` is set to be negative, which makes the buffer smaller than expected. When this happens, the program may crash as a result of writing to an illegal address or being terminated by an error detection mechanism such as `GLIBC` error detection.

Panalyst client logs the packets recently received by the program. In response to a crash, the client identifies the packet being processed and notifies Panalyst server of the error. The server then starts analyzing the vulnerable program at instruction level using an empty HTTP request as a taint source. The request is also described by a set of symbols, which the server uses to compute a symbolic expression for the value of every tainted memory location or register. When the execution of the program reaches Line 1 in Figure 2, the values of the first four bytes on the request need to be revealed so as to determine the branch the execution should follow. For this purpose, the server sends the client a question: "$B_1B_2B_3B_4$ = '`POST`'?", where B_j represents the jth byte on the request. The client checks its privacy policies, which defines the maximal number of bits of information allowed to be leaked for individual HTTP field. In this case, the client is permitted to reveal the keyword `POST` that is deemed nonsensitive. The server then fills the empty request with these letters and moves on to the branch determined by the client's answer. The instruction on Line 2 calls `malloc`. The function accesses memory using a pointer built upon the content of `Content-Length`, which is tainted. To enable this memory access, the server sends the symbolic expression of the pointer to the client to query its concrete value. The client's reply allows the server to add more bytes to the request it is working on. Finally, the execution hits Line 3, a loop to move request content to the buffer allocated through `malloc`. The loop is identified by the server from its repeated instruction pattern. Then, a question is delivered to the client to query its exit condition: " where is the first byte B_j = '$\backslash n$'?". This question concerns request content, a field on which the privacy poli-

cies forbid the client to leak out more than certain amount of information. Suppose that threshold is 5 bytes. To answer the question, only one byte needs to be given away: the position of the byte '$\backslash n$'. Therefore, the client answers the question, which enables the server to construct a new packet to reproduce the crash.

The performance of an analysis can be improved by sending the server an initial report with all the fields that are deemed nonsensitive according the user's privacy policies. In the example, these fields include keywords such as '`POST`' and the `Content-Length` field. This treatment reduces the communication overheads during an analysis.

Threat model. We assume that the user trusts the information provided by the server but does not trust her data with the server. The rationale behind this assumption is based upon the following observations. The owners of the server are often software manufacturers, who have little incentive to steal their customers' information. What the user does not trust is the way in which those parties manage her data, as improper management of the data can result in leaks of her private information. Actually, the same issue is also of concern to those owners, as they could be reluctant to take the liability for protecting user information. Therefore, the client can view the server as a benign though unreliable partner, and take advantage of the information it discovers from the vulnerable program to identify sensitive data, which we elaborate in Section 3.2.

Note that this assumption *is not* fundamental to Panalyst: more often than not, the client is capable of identifying sensitive data on its own. As an example, the aforementioned analysis on the program in Figure 2 does not rely on any trust in the server. Actually, the assumption only serves an approach for defining fine-grained privacy policies in our research (Section 3.2), and elimination of the assumption, though may lead to coarser-grained policies under some circumstances, will not invalidate the whole approach.

3.2 Panalyst Client

Panalyst client is designed to work on the computing devices with various resource constraints. Therefore, it needs to be extremely lightweight. The client also includes a set of policies for protecting the user's privacy and a mechanism to enforce them. We elaborate such a design as follows.

Packet logging and error reporting. Panalyst client intercepts the packets received by an application, extracts their application-level payloads and saves them to a log file. This can be achieved either through capturing packets at network layer using a sniffer such as Wireshark [1], or by interposing on the calls for receiving data from network. We chose the latter for prototyping the client: in our implementation, an application's socket calls are intercepted using `ptrace` [10] to dump the application-level data to a log. The size of the file is bounded, and therefore only the most recent packets are kept.

When a serious runtime error happens, the process of a vulnerable program may crash, which triggers our error analysis mechanism. Runtime errors can also be detected by the mechanisms such as `GLIBC` error detection, Windows build-in diagnostics [11] or other runtime error detection techniques [28, 21]. Once an error happens to an application, Panalyst client identifies the packets it is working on. This is achieved in our design by looking at all the packets within one TCP connection. Specifically, the client marks the beginning of a connection once observing an `accept` call from the application and the end of the connection when it detects `close`. After an exception happens, the client concatenates the application-level payloads of all the packets within the current connection to form a *message*, which it uses to talk to the server. For simplicity, our current design focuses on the error triggered by network input and assumes that all information related to the exploit is present in a single connection. Panalyst can be extended to handle the errors caused by other inputs such as data from a local file through logging and analyzing these inputs. It could also work on multiple connections with the support of the state-of-art replay techniques [43, 32] that are capable of replaying the whole application-layer session to the vulnerable application on the server side. When a runtime error occurs, Panalyst client notifies the server of the type of the error, for example, segmentation fault and illegal instruction. Moreover, the client can ship to the server part of the message responsible for the error, given such information is deemed nonsensitive according to the user's privacy policies.

After reporting to the server a runtime error, Panalyst client starts listening to a port to wait for the questions from the server. Panalyst server may ask two types of questions, either related to a tainted branching condition or a tainted pointer a vulnerable program uses to access memory. In the first case, the client is supposed to answer "yes" or "no" to the question described by a symbolic inequality: $C(B_{k[1]}, \ldots, B_{k[m]}) \leq 0$, where $B_{k[j]}$ ($1 \leq j \leq m$) is the symbol for the $k[j]$th byte on the causal message. In the second case, the client is queried about the concrete value of a symbolic pointer $S(B_{k[1]}, \ldots, B_{k[m]})$. These questions can be easily addressed by the client using the values of these bytes on the message. However, the answers can be delivered to the server only after they are checked against the user's privacy policies, which we describe below.

Privacy policies. Privacy policies here are designed to specify the maximal amount of information that can be given away during an error analysis. Therefore, they must be built upon a proper measure of information. Here, we adopt *entropy* [48], a classic concept of information theory, as the measure. Entropy quantifies uncertainty as number of bits. Specifically, suppose that an application field A is equally likely to take one of m different values. The entropy of A is computed as $\log_2 m$ bits. If the client reveals that A makes a path condition true, which reduces the possible values the field can have to a proportion ρ of m, the exposed information is quantified as: $\log_2 m - \log_2 \rho m = -\log_2 \rho$ bits.

The privacy policies used in Panalyst define the maximal number of bytes of the information within a protocol field that can be leaked out. The number here is called *leakage threshold*. Formally, denote the leakage threshold for a field A by τ. Suppose the server can infer from the client's answers that A can take a proportion ρ of all possible values of that field. The privacy policy requires that the following hold: $-\log_2 \rho \leq \tau$. For example, a policy can specify that no more than 2 bytes of the URL information within an HTTP request can be revealed to the server. This policy design can achieve a fine-grained control of information. As an example, let us consider HTTP requests: protocol keywords such as `GET` and `POST` are usually deemed nonsensitive, and therefore can be directly revealed to the server; on the other hand, the URL field and the cookie field can be sensitive, and need to be protected by low leakage thresholds. Panalyst client includes a protocol parser to partition a protocol message into fields. The parser does not need to be precise: if it cannot tell two fields apart, it just treats them as a single field.

A problem here is that applications may use closed protocols such as ICQ and SMB whose specifications are not publically available. For these protocols, the whole protocol message has to be treated as a single field, which unfortunately greatly reduces the granularity of control privacy policies can have. A solution to this problem is to partition information using the parameters of API (such as Linux kernel API, `GLIBC` or Windows API) functions that work on network input. For example, suppose that the `GLIBC` function `fopen` builds its parameters upon an input message; we can infer that the part of the message related to file access modes (such as '`read`' and '`write`') can be less sensitive than that concerning file name. This approach needs a model of API functions and trust in the information provided by the server. Another solution is to partition an input stream using a set of tokens and common delimiters such as '`\n`'. Such tokens can be specified by the user. For example, using the token '`secret`' and the delimiter '`.`', we can divide the URL '`www.secretservice.gov`' into the following fields: '`www`', '`.`', '`secretservice`' and '`gov`'. Upon these fields, different leakage thresholds can be defined. These two approaches can work together and also be applied to specify finer-grained policies within a protocol field when the protocol is public.

To facilitate specification of the privacy policies, Panalyst can provide the user with policy templates set by the expert. Such an expert can be any party who has the knowledge about fields and the amount of information that can be disclosed without endangering the content of a field. For example, people knowledgeable about the HTTP specifications are in the position to label the fields like '`www`' as nonsensive and domain names such as '`secretservice.gov`' as sensitive. Typically, protocol keywords, delimiters and some API parameters can be treated as public information, while the fields such as those including the tokens and other API parameters are deemed sensitive. A default leakage threshold for a sensitive field can be just a few bytes: for example, we can allow one or two bytes to be disclosed from a domain-name field, because they are too general to be used to pinpoint the domain name; as another example, up to four bytes can be exposed from a field that may involve credit-card numbers, because people usually tolerate such information leaks in real life. Note that we may not be able to assign a zero threshold to a sensitive field because this can easily cause an analysis to fail: to proceed with an analysis, the server often needs to know whether the field contains some special byte such as a delimiter, which gives away a small amount of information regarding its content. These policy templates can be adjusted by a user to define her customized policies.

Policy enforcement. To enforce privacy policies, we need to quantify the information leaked by the client's answers. This is straightforward in some cases but less so in others. For example, we know that answering 'yes' to the question "$B_1B_2B_3B_4$ = 'POST'?" in Figure 2 gives away four bytes; however, information leaks can be more difficult to gauge when it comes to the questions like "$B_j \times B_k < 256?$ ", where B_j and B_k indicates the jth and the kth bytes on a message respectively. Without loss of generality, let us consider a set of bytes $(B_{k[1]}, \ldots, B_{k[m]})$ of a protocol message, whose concrete values on the message makes a condition "$C(B_{k[1]}, \ldots, B_{k[m]}) \leq 0$" true. To quantify the information an answer to the question gives away, we need to know ρ, the proportion of all possible values these bytes can take that make the condition true. Finding ρ is nontrivial because the set of the values these bytes can have can be very large, which makes it impractical to check them one by one against the inequality. Our solution to the problem is based upon the classic statistic technique for estimating a proportion in a popu-

lation. Specifically, we randomly pick up a set of values for these bytes to verify a branching condition and repeat the trial for n times. From these n trials, we can estimate the proportion ρ as $\frac{x}{n}$ where x is the number of trials in which the condition is true. The accuracy of this estimate is described by the probability that a range of values contain the true value of ρ. The range here is called *confidence interval* and the probability called *confidence level*. Given a confidence interval and a confidence level, standard statistic technique can be used to determine the size of samples n [2]. For example, suppose the estimate of ρ is 0.3 with a confidence interval ± 0.5 and a confidence level 0.95, which intuitively means $0.25 < \rho < 0.35$ with a probability 0.95; in this case, the number of trials we need to play is 323. This approach offers an approximation of information leaks: in the prior example, we know that with 0.95 confidence, information being leaked will be no more than $-\log_2 0.25 = 4$ bits. Using such an estimate and a predetermined leakage threshold, a policy enforcer can decide whether to let the client answer a question.

3.3 Panalyst Server

Panalyst server starts working on a vulnerable application upon receiving an initial error report from the client. The report includes the type of the error, and other non-sensitive information such as the corrupted pointer, the lengths of individual packets' application-level payloads and the content of public fields. Based upon it, the server conducts an instruction-level analysis of the application's executable, which we elaborate as follows.

Taint analysis and symbolic execution. Panalyst server performs a dynamic taint analysis on the vulnerable program, using a network input built upon the initial report as a taint source. The input involves a set of packets, whose application-layer payloads form a message characterized by the same length as the client's message and the information disclosed by the report. The server monitors the execution of the program instruction by instruction to track tainted data according to a set of taint-propagation rules. These rules are similar to those used in other taint-analysis techniques such as RIFLE [51], TaintCheck [44] and LIFT [45], examples of which are presented in Table 1. Along with the dynamic analysis, the server also performs a symbolic execution [37] that statically evaluates the execution of the program through interpreting its instructions, using symbols instead of real values as input. Each symbol used by Panalyst represents one byte on the input message. Analyzing the program in this way, we can not only keep close track of tainted data flows, but also formulate a symbolic expression for every tainted value in memory and registers.

Whenever the execution encounters a conditional branching with its condition tainted by input symbols, the server sends the condition as a question to the client to seek answer. With the answer from the client, the server can find *hypothetic values* for these symbols using a constraint solver. For example, a "no" to the question $B_i =$ '$\backslash n$' may result in a letter 'a' to be assigned to the ith byte on the input. To keep the runtime data consistent with the hypothetic value of symbol B_i, the server updates all the tainted values related to B_i by evaluating their symbolic expressions with the hypothetic value. It is important to note that B_i may appear in multiple branching conditions ($C_1 \leq 0$, ..., $C_k \leq 0$). Without loss of generality, suppose all of them are true. To find a value for B_i, the constraint solver must solve the constraint $(C_1 \leq 0) \wedge \ldots \wedge (C_k \leq 0)$. The server also needs to "refresh" the tainted values concerning B_i each time when a new hypothetic value of the symbol comes up.

The server also queries the client when the program attempts to access memory through a pointer tainted by input symbols $(B_{k[1]}, \ldots, B_{k[m]})$. In this case, the server needs to give the symbolic expression of the pointer $S(B_{k[1]}, \ldots, B_{k[m]})$ to the client to get its value v, and solve the constraint $S(B_{k[1]}, \ldots, B_{k[m]}) = v$ to find these symbols' hypothetic values. Query of a tainted pointer is necessary for ensuring the program's correct execution, particularly when a write happens through such a pointer. It is also an important step for reliably reproducing a runtime error, as the server may need to know the value of a pointer, or at least its range, to determine whether an illegal memory access is about to occur. However, this treatment may disclose too much user information, in particular when the pointer involves only one symbol: a "yes" to such a question often exposes the real value of that symbol. Such a problem usually happens in a string-related `GLIBC` function, where letters on a string are used as offsets to look up a table. Our solution is to accommodate symbolic pointers in our analysis if such a pointer carries only one symbol and is used to read from a memory location. This approach can be explicated through an example. Consider the instruction "`MOV EAX, [ESI+CL]`", where `CL` is tainted by an input byte B_j. Instead of directly asking the client for the value of `ESI+CL`, which reveals the real value of B_j, the server gathers the bytes from the memory locations pointed by (`ESI+0`, `ESI+1`, ..., `ESI+ 255`) to form a list. The list is used to prepare a question should `EAX` get involved in a branching condition such as "`CMP EAX, 1`". In this case, the server generates a query including `[ESI+CL]`, which is the symbolic expression of `EAX`, the value of `ESI`, the list and the condition. In response to the query, the client uses the real value of B_j and the list to verify the condition and answer either "yes" or "no", which enables the server to identify the right branch.

Table 1: Examples of the Taint Rules.

Instruction Category	Taint Propagation	Examples
data movement	(1) taint is propagated to the destination if the source is tainted, (2) the destination operand is not tainted if the source operand is not tainted.	`mov eax,ebx;` `push eax;` `call 0x4080022;` `lea ebx, ptr [ecx+10]`
arithmetic	(1) taint is propagated to the destination if the source is tainted, (2) the EFLAGS is also regarded as a destination operand.	`and eax, ebx;` `inc ecx;` `shr eax,0x8`
address calculation	an address is tainted if any element in the address calculation is tainted	`mov ebx, dword ptr [ecx+2*ebx+0x08]`
conditional jump	regard EFLAGS as a source operand	`jz 0x0746323;` `jnle 0x878342;` `jg 0x405687`
compare	regard EFLAGS as a destination operand	`cmp eax,ebx;` `test eax,eax`

The analysis stops when the execution reaches a state where a runtime error is about to happen. Examples of such a state include a jump to an address outside the process image or an illegal instruction, and memory access through an illegal pointer. When this happens, Panalyst server announces that an input reproducing the error has been identified, and can be used for further analysis of the underlying bug and generation of signatures [52, 50, 39] or patches [49]. Our analysis also contributes to a preliminary classification of bugs: if the illegal address that causes the error is found to be tainted, we have a reason to believe that the underlying bug can be exploited remotely and therefore is security critical.

Reducing communication overhead. A major concern for Panalyst seems to be communication overhead: the server may need to query the client whenever a tainted branching condition or a tainted pointer is encountered. However, in our research, we found that the bandwidth consumed in an analysis usually is quite small, less than a hundred KB during the whole analysis. This is because the number of tainted conditions and pointers can be relatively small in many programs, and both the server's questions and the client's answers are usually short. Need for communication can be further reduced if an initial error report supplies the server with a sufficient amount of public information regarding the error. However, the performance of the server and the client will still be affected when the program intensively operates on tainted data, which in many cases is related to loop.

A typical loop that appears in many network-facing applications is similar to the one in the example (Line 6 of Figure 2). The loop compares individual bytes in a protocol field with a delimiter such as '$\backslash n$' or ' ' to identify the end of the field. If we simply view the loop as a sequence of conditional branching, then the server has to query the client for every byte within that field, which can be time consuming. To mitigate this problem, we designed a technique in our research to first identify such a loop and then let client proactively scan its message to find the location of the first string that terminates the loop. We describe the technique below.

The server monitors a tainted conditional branching that the execution has repeatedly bumped into. When the number of such encounters exceeds a threshold, we believe that a loop has been identified. The step value of that loop can be approximated by the difference between the indices of the symbols that appear in two consecutive evaluations of the condition. For example, consider the loop in Figure 2. If the first time the execution compares B_j with '$\backslash n$' and the second time it tries B_{j+1}, we estimate the step as one. The server then sends a question to the client, including the loop condition $C(B_{k[1]}, \ldots, B_{k[m]})$ and step estimates $\lambda_{k[1]}, \ldots, \lambda_{k[m]}$. The client starts from the $k[i]$th byte ($1 \leq i \leq m$) to scan its message every $\lambda_{k[j]}$ bytes, until it finds a set of bytes $(B'_{k[1]}, \ldots, B'_{k[m]})$ that makes the condition false. The positions of these bytes are shipped to the server. As a result, the analysis can evaluate the loop condition using such information, without talking to the client iteration by iteration.

The above technique only works on a simple loop characterized by a constant step value. Since such a loop frequently appears in network-facing applications, our approach contributes to significant reduction of communication when analyzing these applications. Development of a more general approach for dealing with the loops with varying step size is left as our future research. Another problem of our technique is that the condition it identifies may not be a real loop condition. However, this does not bring us much trouble in general, as the penalty of such a false positive can be small, including nothing but the requirement for the client to scan its message and disclosure of a few bytes that seem to meet the exit condition. If the client refuses to do so, the analysis can still continue through directly querying the client about branching conditions.

Improving constraint-solving performance. Solving a constraint can be time consuming, particularly when the constraint is nonlinear, involving operations such as bitwise AND, OR and XOR. To maintain a valid runtime state for the program under analysis, Panalyst server

needs to run a constraint solver to update hypothetic symbol values whenever a new branching condition or memory access is encountered. This will impact the server's performance. In our research, we adopted a very simple strategy to mitigate this impact: we check whether current hypothetic values satisfy a new constraint before solving the constraint. This turns out to be very effective: in many cases, we found that symbol values good for an old constraint also work for a new constraint, which allows us to skip the constraint-solving step.

4 Implementation

We implemented a prototype of Panalyst under Linux, including its server component and client component. The details of our implementation are described in this section.

Message logging. We adopted `ptrace` to dump the packet payloads an application receives. Specifically, `ptrace` intercepts the system call `socketcall()` and parses its parameters to identify the location of an input buffer. The content of the buffer is dumped to a log file. We also labels the beginning of a connection when an `accept()` is observed and the end of the connection when there is a `close()`. The data between these two calls are used to build a message once a runtime exception happens to the application.

Estimate of information leaks. To evaluate the information leaks caused by answering a question, our implementation first generates a constraint that is a conjunction of all the constraints the client receives that are directly or transitively related to the question, and then samples values of the constraint using the random values of the symbols it contains. We set the number of samples to 400, which achieves a confidence interval of ± 0.05 and a confidence level of 0.95. A problem here is that the granularity of the control here could be coarse, as 400 samples can only represent loss of one byte of information. When this happens, our current implementation takes a conservative treatment to assume that all the bytes in a constraint are revealed. A finer-grained approach can be restoring the values of the symbols byte by byte to repeatedly check information leaks, until all the bytes are disclosed. An evaluation of such an approach is left as our future work.

Error analyzer. We implemented an error analyzer as a Pin tool that works under Pin's Just-In-Time (JIT) mode [40]. The analyzer performs both taint analysis and symbolic execution on a vulnerable application, and builds a new input to reproduce the runtime error that occurred on the client. The analyzer starts from a message that contains nothing but zeros and has the same length as the client's input, and designates a symbol to every byte on that message. During the analysis, the analyzer first checks whether a taint will be propagated by an instruction and only symbolically evaluates those whose operands involve tainted bytes. Since many instructions related to taint propagation use the information of `EFLAGS`, the analyzer also takes this register as a source operand for these instructions. Once an instruction's source operand is tainted, symbolic expressions are computed for the destination operand(s). For example, consider the instruction `add eax, ebx`, where `ebx` is tainted. Our analyzer first computes a symbolic expression $B_{ebx} + v_{eax}$, where B_{ebx} is an expression for `ebx` and v_{eax} is the value of `eax`, and then generates another expression for `EFLAGS` because the result of the operation affects Flag `OF, SF, ZF, AF, CF, PF`.

Whenever a conditional jump is encountered, the server queries the client about `EFLAGS`. To avoid asking the client to give away too much information, such a query only concerns the specific flag that affects that branching, instead of the whole status of `EFLAGS`. For example, consider the following branching: `cmp eax,ebx` and then `jz 0x33fd740`. In this case, the server's question is only limited to the status of `ZF`, which the branching condition depends on, though the comparison instruction also changes other flags such as `SF` and `CF`.

Constraint solver. Our implementation uses Yices [33] to solve constraints so as to find the hypothetic values for individual symbols. These values are important to keeping the application in a state that is consistent with its input. Yices is a powerful constraint solver which can handle many nonlinear constraints. However, there are situations when a constraint is so complicated that its solution cannot be obtained within a reasonable time. When this happens, we adopted a strategy that gradually inquires the client about the values of individual symbols to simplify the constraint, until it becomes solvable by the constraint solver.

Data compression. We implemented two measures to reduce the communication between the client and the server. The first one is for processing the questions that include the same constraints except input symbols. Our implementation indexes each question the server sends to the client. Whenever the server is about to ask a question that differs from a previous one only in symbols, it only transmits the index of the old question and these symbols. This strategy is found to be extremely effective when the sizes of the questions become large: in our experiment, a question with 8KB was compressed to 52 bytes. The strategy also complements our technique for processing loops: for a complicated loop with varying steps which the technique cannot handle, the server

needs to query the client iteratively; however, the sizes of these queries can be very small as they are all about the same constraint with different symbols. The second measure is to use a lightweight real-time compression algorithm to reduce packet sizes. The algorithm we adopted is minilzo [6], which reduced the bandwidth consumption in our experiments to less than 100 KB for an analysis, at a negligible computational overhead.

5 Evaluation

In this section, we describe our experimental study of Panalyst. The objective of this study is to understand the effectiveness of our technique in remote error analysis and protection of the user's privacy, and the overheads it introduces. To this end, we evaluated our prototype using 6 real applications and report the outcomes of these experiments here.

Our experiments were carried out on two Linux workstations, one as the server and the other as the client. Both of them were installed with Redhat Enterprise 4. The server has a 2.40GHz Core 2 Duo processor and 3GB memory. The client has a Pentium 4 1.3GHz processor and 256MB memory.

5.1 Effectiveness

We ran Panalyst to analyze the errors that occurred in 6 real applications, including Newspost [7], OpenVMPS [19], Null-HTTPd (Nullhttpd) [8], Sumus [15], Light HTTPd [5] and ATP-HTTPd [3]. The experimental results are presented in Table 2. These applications contain bugs that are subject to stack-based overflow, format string error and heap-based overflow. The errors were triggered by a single or multiple input packets on the client and analyzed on the server. As a result, new packets were gradually built from an initial error report and interactions with the client to reproduce an error. This was achieved without leaking too much user information. We elaborate our experiments below.

Newspost. Newspost is a Usenet binary autoposter for Unix and Linux. Its version 2.1.1 and earlier has a bug subject to stack-based overflow: specifically, a buffer in the `socket_getline()` function can be overrun by a long string without a newline character. In our experiment, the application was crashed by a packet of 2KB. After this happened, the client sent the server an initial error report that described the length of the packet and the type of the error. The report was converted into an input to an analysis performed on the application, which included an all-zero string of 2KB. During the analysis, the server identified a loop that iteratively searched for '`0xa`', the newline symbol, as a termination condition for moving bytes into a buffer, and questioned the client about the position at which the byte first appeared. The byte actually did not exist in the client's packet. As a result, the input string overflowed the buffer and was spilled on an illegal address to cause a segmentation fault. Therefore, the server's input was shown to be able to reproduce the error. This analysis was also found to disclose very little user information: nothing more than the fact that none of the input bytes were '`0xa`' were revealed. This was quantified as 0.9 byte.

OpenVMPS. OpenVMPS is an open-source implementation of Cisco Virtual Membership Policy Server, which dynamically assigns ports to virtual networks according to Ethernet addresses. The application has a format string bug which allows the input to supply a string with format specifiers as a parameter for `vfprintf()`. This could make `vfprintf()` write to a memory location. In the experiment, Panalyst server queried the client to get "`00 00 0c 02`" as illustrated in Figure 4. These four bytes were part of a branching condition, and seems to be a keyword of the protocol. We also found that the string "`00 b9`" were used as a loop counter. These two bytes were identified by the constraint solver. The string "`62637`" turned out to be the content that the format specifier "`%19$hn`" wrote to a memory location through `vfprintf()`. They were recovered from the client because they were used as part of a pointer to access memory. Our implementation successfully built a new input on the server that reproduced the error, as illustrated in Figure 4. This analysis recovered 39 bytes from the client, all of which were either related to branching conditions or memory access. An additional 18.4 bytes of information were estimated by the client to be leaked, as a result of the client's answers which reduced the ranges of the values some symbols could take.

Null-HTTPd. Null-HTTPd is a small web server working on Linux and Windows. Its version 0.5 contains a heap-overflow bug, which can be triggered when the HTTP request is a `POST` with a negative `Content Length` field and a long request content. In our experiment, the client parsed the request using Wireshark and delivered nonsensitive information such as the keyword `POST` to the server. The server found that the application added 1024 to the value derived from the `Content Length` and used the sum as pointer in the function `calloc`. This resulted in a query for the value of that field, which the client released. At this point, the server acquired all the information necessary for reproducing the error and generated a new input illustrated in Figure 5. The information leaks caused by the analysis include the keyword, the value of `Content Length`, HTTP delimiters and the knowledge that some bytes are not special symbols such as delimiters. This was quan-

Table 2: Effectiveness of Panalyst.

Applications	Vul. Type	New Input Generated?	Size of client's message (bytes)	Info leaks (bytes)	Rate of info leaks
Newspost	Stack Overflow	Yes	2056	0.9	0.04%
OpenVMPS	Format String	Yes	199	57.4	28.8%
Null-HTTPd	Heap Overflow	Yes	416	29.7	7.14%
Sumus	Stack Overflow	Yes	500	7.7	1.54%
Light HTTPd	Stack Overflow	Yes	211	17.9	8.48%
ATP-HTTPd	Stack Overflow	Yes	819	16.7	2.04%

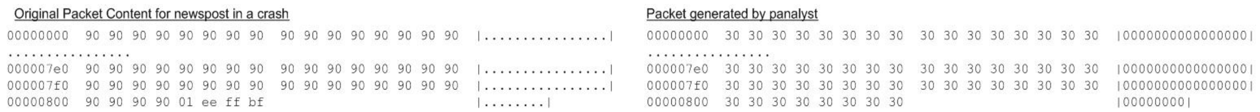

Figure 3: Input Generation for Newspost. Left: the client's packet; Right: the new packet generated on the server.

tified as 29.7 bytes, about 7% of the HTTP message the client received.

Sumus. Sumus is a server for playing Spanish "mus" game on the Internet. It is known that Sumus 0.2.2 and the earlier versions have a vulnerable buffer that can be overflowed remotely [14]. In our experiment, Panalyst server gradually constructed a new input through interactions with the client until the application was found to jump to a tainted address. At this point, the input was shown to be able to reproduce the client's error. The information leaked during the analysis is presented in Figure 6, including a string "GET" which affected a path condition, and 4 "`0x90`", which were the address the application attempted to access. These 7 bytes were counted as leaked information, along with the fact that other bytes were not a delimiter.

Light-HTTPd. Light-HTTPd is a free HTTP server. Its version 0.1 has a vulnerable buffer on the stack. Our experiment captured an exception that happened when the application returned from the function `vsprintf()` and constructed the new input. The input shared 14 bytes with the client's input which were essential to determining branching conditions and accessing memory. For example, the keyword "`GET`" appeared on a conditional jump and the letter "`H`" were used as a condition in the `GLIBC` function `strstr`. The remaining 3.9 bytes were caused by the intensive string operations, such as `strtok`, which frequently used individual bytes for table lookup and comparison operations. Though these operations did not give away the real values of these bytes, they reduced the range of the bytes, which were quantified into another 3.9 bytes.

ATP-HTTPd. ATP-HTTPd 0.4 and 0.4b involve a remotely exploitable buffer in the `socket_gets()` function. A new input that triggered this bug was built in our experiment, which are presented in Figure 8. For example, the string "`EDCB`" was an address the application attempted to jump to; this operation actually caused a segmentation fault. Information leaks during this analysis are similar to that of Light-HTTPd, which was quantified as 16.7 bytes.

5.2 Performance

We also evaluated the performance of Panalyst. The client was deliberately run on a computer with 1 GHz CPU and 256MB memory to understand the performance impact of our technique on a low-end system. The server was on a high-end, with a 2.40GHz Core 2 Duo CPU and 3GB memory. In our experiments, we measured the delay caused by an analysis, memory use and bandwidth consumption on both the client and the server. The results are presented in Table 3.

The client's delay describes the accumulated time that the client spent to receive packets from the server, compute answers, evaluate information leaks and deliver the responses. In our experiments, we observed that this whole process incurred the latency below 3.2 seconds. Moreover, the memory use on the client side was kept below 5 MB. Given the hardware platform over which this performance was achieved, we have a reason to believe that such overhead could be afforded by even a device with limited computing resources, such as Pocket PC and PDA. Our analysis introduced a maximal 99,659 bytes communication overhead. We believe this is still reasonable for the client, because the size of a typical web page exceeds 100 KB and many mobile devices nowadays have the capability of web browsing.

The delay on the server side was measured between the reception of an initial error report and the generation of a new input. An additional 15 seconds for launching our Pin-based analyzer should also be counted. Given this, the server's performance was very good: the maximal latency was found to be under 1 minute. However,

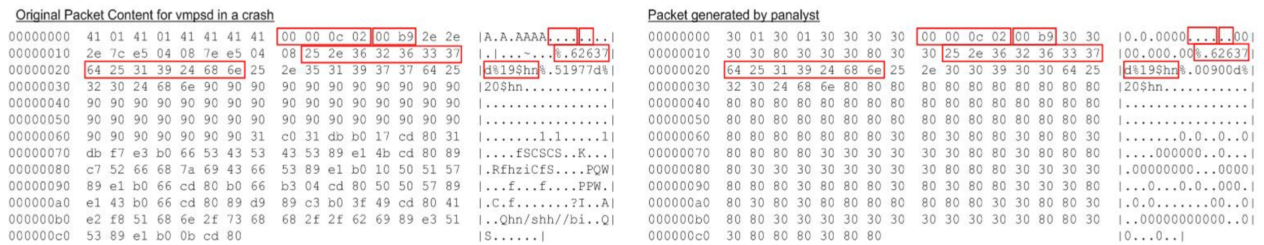

Figure 4: Input Generation for OpenVMPS. Left: the client's packet; Right: the new packet generated on the server.

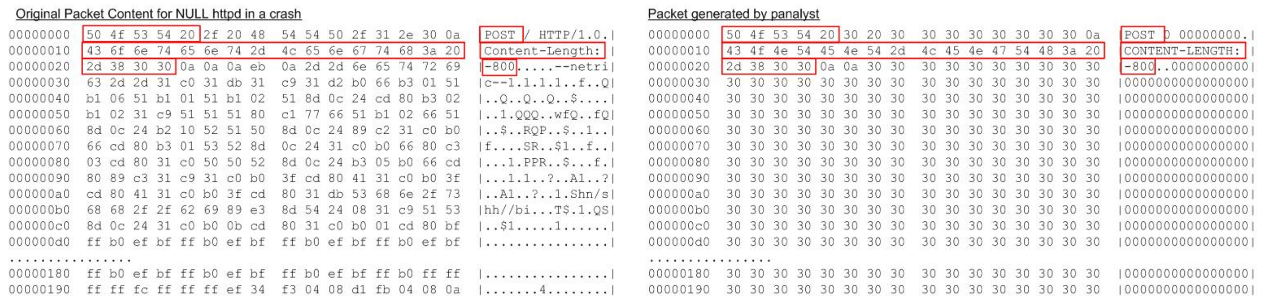

Figure 5: Input Generation for Null-HTTPd. Left: the client's packet; Right: the new packet generated on the server.

this was achieved on a very high-end system. Actually, we observed that the latency was doubled when moving the server to a computer with 2.36 GHz CPU and 1 GB memory. More importantly, the server consumed about 100 MB memory during the analysis. This can be easily afforded by a high-end system as the one used in our experiment, but could be a significant burden to a low-end system such as a mobile device. As an example, most PDAs have less than 100 MB memory. Therefore, we believe that Panalyst server should be kept on a dedicated high-performance system.

6 Discussion

Our research makes the first step towards a fully automated and privacy-aware remote error analysis. However, the current design of Panalyst is still preliminary, leaving much to be desired. For example, the approach does not work well in the presence of probabilistic errors, and our privacy policies can also be better designed. We elaborate limitations and possible solutions in the left part of this section, and discuss the future research for improving our technique in Section 7.

The current design of Panalyst is for analyzing the error triggered by network input alone. However, runtime errors can be caused by other inputs such as those from a local file or another process. Some of these errors can also be handled by Panalyst. For example, we can record all the data read by a vulnerable program and organize them into multiple messages, each of which corresponds to a particular input to the program; an error analysis can happen on these messages in a similar fashion as described in Section 3. A weakness of our technique is that it can be less effective in dealing with a probabilistic error such as the one caused by multithread interactions. However, it can still help the server build sanitized inputs that drive the vulnerable program down the same execution paths as those were followed on the client.

Panalyst may require the client to leak out some information that turns out to be unnecessary for reproducing an error, in particular, the values of some tainted pointer unrelated to the error. A general solution is describing memory addresses as symbolic expressions and taking them into consideration during symbolic execution. This approach, however, can be very expensive, especially when an execution involves a large amount of indirect addressing through the tainted pointers. To maintain a moderate overhead during an analysis, our current design only offers a limited support for symbolic pointers: we introduce such a pointer only when it includes a single symbol and is used for reading from memory.

The way we treat loops is still preliminary: it only works on the loops with constant step sizes and may falsely classify a branching condition as a loop condition. As a result, we may miss some real loops, which increases the communication overhead of an analysis, or require the client to unnecessarily disclose extra informa-

Original Packet Content for sumus in a crash

```
00000000  08 08 08 08 08 08 08 08  08 08 08 08 08 08 08 08  |................|
00000010  08 08 08 08 08 08 08 08  08 08 08 08 08 08 08 08  |................|
00000020  08 08 08 08 08 08 08 08  08 08 08 08 08 08 08 08  |................|
00000030  08 08 08 08 08 08 08 08  08 08 08 08 08 08 08 08  |................|
00000040  08 08 08 08 08 08 08 08  08 08 08 08 08 08 08 08  |................|
00000050  08 08 08 08 08 08 47 45  54 08 08 08 08 08 08 08  |......GET.......|
00000060  08 08 08 08 08 08 08 08  08 08 08 08 08 08 08 08  |................|
................
000000b0  08 08 08 08 08 08 08 08  08 08 08 08 08 08 08 08  |................|
000000c0  08 08 08 73 01 a0 05 08  90 90 90 90 90 90 90 90  |...s............|
000000d0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
................
00000180  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
00000190  90 90 90 90 90 90 90 90  31 c0 50 40 89 c3 50 40  |........1.P@..P@|
000001a0  50 89 e1 b0 66 cd 80 31  d2 52 66 68 1f 2b 43 66  |P...f..1.Rfh.+Cf|
000001b0  53 89 e1 6a 10 51 50 89  e1 b0 66 cd 80 40 89 44  |S..j.QP...f..@.D|
000001c0  24 04 43 43 b0 66 cd 80  83 c4 0c 52 52 43 b0 66  |$.CC.f.....RRC.f|
000001d0  cd 80 93 89 d1 b0 3f cd  80 41 80 f9 03 75 f6 52  |......?..A...u.R|
000001e0  68 6e 2f 73 68 68 2f 2f  62 69 89 e3 52 53 89 e1  |hn/shh//bi..RS..|
000001f0  b0 0b cd 80                                       |....|
```

Packet generated by panalyst

```
00000000  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000010  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000020  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000030  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000040  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000050  30 30 30 30 30 30 47 45  54 30 30 30 30 30 30 30  |000000GET0000000|
00000060  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
................
000000b0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000000c0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000000d0  30 30 30 90 90 90 90 90  30 30 30 30 30 30 30 30  |000.....00000000|
................
00000180  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000190  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000001a0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000001b0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000001c0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000001d0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000001e0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000001f0  30 30 30 30                                       |0000|
```

Figure 6: Input Generation for Sumus. Left: the client's packet; Right: the new packet generated on the server.

Original Packet Content for light httpd in a crash

```
00000000  47 45 54 20 2f 5e 5e 90  90 90 90 90 90 90 90 90  |GET /^^.........|
00000010  90 90 90 eb 72 5e 29 c0  89 46 10 40 89 c3 89 46  |....r^)..F.@...F|
00000020  0c 40 89 46 08 8d 4e 08  b0 66 cd 80 43 c6 46 10  |.@.F..N..f..C.F.|
00000030  10 66 89 5e 14 88 46 08  29 c0 89 c2 89 46 18 b0  |.f.^..F.)....F..|
00000040  90 66 89 46 16 8d 4e 14  89 4e 0c 8d 4e 08 b0 66  |.f.F..N..N..N..f|
00000050  cd 80 89 5e 0c 43 43 b0  66 cd 80 89 56 0c 89 56  |...^.CC.f...V..V|
00000060  10 b0 66 43 cd 80 86 c3  b0 3f 29 c9 cd 80 b0 3f  |..fC.....?)....?|
00000070  41 cd 80 b0 3f 41 cd 80  88 56 07 89 76 0c 87 f3  |A...?A...V..v...|
00000080  8d 4b 0c b0 0b cd 80 e8  89 ff ff ff 2f 62 69 6e  |.K........../bin|
00000090  2f 73 68 a0 b5 ff bf a0  b5 ff bf a0 b5 ff bf a0  |/sh.............|
000000a0  b5 ff bf a0 b5 ff bf a0  b5 ff bf a0 b5 ff bf a0  |................|
000000b0  b5 ff bf a0 b5 ff bf a0  b5 ff bf a0 b5 ff bf a0  |................|
000000c0  b5 ff bf a0 b5 ff bf 20  48 54 54 50 2f 31 2e 30  |....... HTTP/1.0|
000000d0  0d 0a 0a                                          |...|
```

Packet generated by panalyst

```
00000000  47 45 54 20 2f 30 30 30  30 30 30 30 30 30 30 30  |GET /00000000000|
00000010  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000020  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000030  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000040  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000050  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000060  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000070  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000080  30 30 30 30 30 30 30 30  30 30 30 30 2f 30 30 30  |000000000000/000|
00000090  2f 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |/000000000000000|
000000a0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000000b0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000000c0  30 30 30 30 30 30 bf 20  48 30 30 30 2f 30 30 30  |000000. H000/000|
000000d0  0d 0a 0a                                          |...|
```

Figure 7: Input Generation for Light HTTPd. Left: the client's packet; Right: the new packet generated on the server.

tion. However, the client can always refuse to give more information and set a threshold for the maximal number of the questions it will answer. Even if this causes the analysis to fail, the server can still acquire some information related to the error and use it to facilitate other error analysis techniques such as fuzz testing. We plan to study more general techniques for analyzing loops in our future research.

Entropy-based policies may not be sufficient for regulating information leaks. For example, complete disclosure of one byte in a field may have different privacy implications from leakage of the same amount of information distributed among several bytes in the field. In addition, specification of such policies does not seem to be intuitive, which may affect their usability. More effective privacy policies can be built upon other definitions of privacy such as k-Anonymity [46], l-Diversity [41] and t-Closeness [38]. These policies will be developed and evaluated in our future work.

Panalyst client can only approximate the amount of information disclosed by its answers using statistical means. It also assumes a uniform distribution over the values a symbol can take. Design of a better alternative for quantifying and controlling information is left as our future research.

Another limitation of our approach is that it cannot handle encoded or encrypted input. This problem can be mitigated by interposing on the API functions (such as those in the OpenSSL library) for decoding or decryption to get their plaintext outputs. Our error analysis will be conducted over the plaintext.

7 Related Work

Error reporting techniques have been widely used for helping the user diagnose application runtime error. Windows error reporting [20], a technique built upon Microsoft's Dr. Watson service [18], generates an error report through summarizing a program state, including contents of registers and stack. It may also ask the user for extra information such as input documents to investigate an error. Such an error report is used to search an expert system for the solution provided by human experts. If the search fails, the client's error will be recorded for a future analysis. Crash Reporter [16] of `Mac OS X` and third-party tools such as BugToaster [27] and Bug Buddy [22] work in a similar way. As an example, Bug Buddy for GNOME can generate a stack trace using `gdb` and let the user post it to the GNOME bugzilla [4].

Privacy protection in existing error reporting techniques mostly relies on the privacy policies of those who collect reports. This requires the user to trust the collector, and also forces her to either send the whole report

Original Packet Content for ATP httpd in a crash

```
00000000  47 45 54 20 2f 41 41 41  41 41 41 41 41 41 41 41  |GET /AAAAAAAAAAA|
00000010  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  |AAAAAAAAAAAAAAAA|
................
00000180  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  |AAAAAAAAAAAAAAAA|
00000190  41 41 41 41 41 41 41 41  41 41 45 44 43 42 45 44  |AAAAAAAAAAEDCBED|
000001a0  43 42 45 44 43 42 45 44  43 42 45 44 43 42 45 44  |CBEDCBEDCBEDCBED|
................
000002c0  43 42 45 44 43 42 45 44  43 42 45 44 43 42 45 44  |CBEDCBEDCBEDCBED|
000002d0  43 42 45 44 43 42 45 44  43 42 45 44 43 42 45 44  |CBEDCBEDCBEDCBED|
000002e0  43 42 45 44 43 42 45 44  43 42 45 44 43 42 45 44  |CBEDCBEDCBEDCBED|
000002f0  43 42 45 44 43 42 45 44  43 42 45 44 43 42 45 44  |CBEDCBEDCBEDCBED|
00000300  43 42 45 44 43 42 45 44  43 42 45 44 43 42 45 44  |CBEDCBEDCBEDCBED|
00000310  43 42 45 44 43 42 45 44  43 42 45 44 43 42 45 44  |CBEDCBEDCBEDCBED|
00000320  43 42 45 44 43 42 45 44  20 48 54 54 50 2f 31 2e  |CBEDCBED HTTP/1.|
00000330  31 0d 0a                                          |1..|
```

Packet generated by panalyst

```
00000000  47 45 54 20 2f 30 30 30  30 30 30 30 30 30 30 30  |GET /00000000000|
00000010  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
................
00000180  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000190  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000001a0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
................
000002c0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000002d0  30 30 30 30 30 30 45 44  43 42 30 30 30 30 30 30  |000000EDCB000000|
000002e0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000002f0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000300  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000310  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000320  30 30 30 30 30 30 30 30  20 30 30 30 30 2f 30 30  |00000000 0000/00|
00000330  30 0d 0a                                          |0..|
```

Figure 8: Input Generation for ATP HTTPd. Left: the client's packet; Right: the new packet generated on the server.

Table 3: Performance of Panalyst.

Programs	client delay (s)	client memory use (MB)	server delay (s)	server memory use (MB)	total size of questions (bytes)	total size of answers (bytes)
Newspost	0.022	4.7	12.14	99.3	527	184
OPenVMPS	1.638	3.9	17	122.3	45,610	6,088
Null-HTTPd	1.517	5.0	13.09	118.1	99,659	3,416
Sumus	0.123	4.8	1.10	85.4	5,968	2,760
Light HTTPd	0.88	4.8	6.59	110.1	14,005	2,808
ATP-HTTPd	3.197	5.0	37.11	145.4	50,615	15,960

or submit nothing at all. In contrast, Panalyst reduces the user's reliance on the collectors to protect her privacy and also allows her to submit part of the information she is comfortable with. Even if such information is insufficient for reproducing an error, it can make it easier for other techniques to identify the underlying bug. Moreover, Panalyst server can automatically analyze the error caused by an unknown bug, whereas existing techniques depend on human to analyze new bugs.

Proposals have been made to improve privacy protection during error reporting. Scrash [25] instruments an application's source code to record information related to a crash and generate a "clean" report that does not contain sensitive information. However, it needs source code and therefore does not work on commodity applications without the manufacturer's support. In addition, the technique introduces performance overheads even when the application works properly, and like other error reporting techniques, uses a remote expert system and therefore does not perform automatic analysis of new errors. Brickell, et al propose a privacy-preserving diagnostic scheme, which works on binary executables [24, 36]. The technique aims at searching a knowledge base framed as a decision tree in a privacy-preserving manner. It also needs to profile an application's execution. Panalyst differs from these approaches in that it does not interfere with an application's normal run except logging inputs, which is very lightweight, and is devised for automatically analyzing an unknown bug.

Techniques for automatic analysis of software vulnerabilities have been intensively studied. Examples include the approach for generating vulnerability-based signatures [26], Vigilante [30], DACODA [31] and EXE [53]. These approaches assume that an input triggering an error is already given and therefore privacy is no longer a concern. Panalyst addresses the important issue on how to get such an input without infringing too much on the user's privacy. This is achieved when Panalyst server is analyzing the vulnerable program. Our technique combines dynamic taint analysis with symbolic execution, which bears some similarity to a recent proposal for exploring multiple execution paths [42]. However, that technique is primarily designed for identifying hidden actions of malware, while Panalyst is for analyzing runtime errors. Therefore, we need to consider the issues that are not addressed by the prior approach. A prominent example is the techniques we propose to tackle a tainted pointer, which is essential to reliably reproducing an error.

Similar to Panalyst, a technique has been proposed recently to symbolically analyze a vulnerable executable and generate an error report through solving constraints [29]. The technique also applies entropy to quantify information loss caused by the error reporting. Panalyst differs from that approach fundamentally in that our technique generates a new input remotely while the prior approach directly works on the causal input on the client. Performing an intensive analysis on the client is exactly the thing we want to avoid, because this increases the client's burden and thus discourages the user from participating. Although an evaluation of the technique reports a moderate overhead [29], it does not include computation-intensive operations such as instruction-level tracing, which can, in some cases, introduce hundreds of seconds of delay and hundreds of megabytes of execution traces [23]. This can be barely

acceptable to the user having such resources, and hardly affordable to those using weak devices such as PocketPC and PDA. Actually, reproducing an error without direct access to the causal input is much more difficult than analyzing the input locally, because it requires a careful coordination between the client and the server to ensure a gradual release of the input information without endangering the user's privacy and failing the analysis at the same time. In addition, Panalyst can enforce privacy policies to individual protocol fields and therefore achieves a finer-grained control of information than the prior approach.

8 Conclusion and Future Work

Remote error analysis is essential to timely discovery of security critical vulnerabilities in applications and generation of fixes. Such an analysis works most effectively when it protects users' privacy, incurs the least performance overheads on the client and provides the server with sufficient information for an effective study of the underlying bugs. To this end, we propose Panalyst, a new techniques for privacy-aware remote error analysis. Whenever a runtime error occurs, the Panalyst client sends the server an initial error report that includes nothing but the public information about the error. Using an input built from the report, Panalyst server analyzes the propagation of tainted data in the vulnerable application and symbolically evaluates its execution. During the analysis, the server queries the client whenever it does not have sufficient information to determine the execution path. The client responds to a question only when the answer does not leak out too much user information. The answer from the client allows the server to adjust the content of the input through symbolic execution and constraint solving. As a result, a new input will be built which includes the necessary information for reproducing the error on the client. Our experimental study of this technique demonstrates that it exposes a very small amount of user information, introduces negligible overheads to the client and enables the server to effectively analyze an error.

The current design of Panalyst is for analyzing the error triggered by network inputs alone. Future research will extend our approach to handle other types of errors. In addition, we also plan to improve the techniques for estimating information leaks and reduce the number of queries the client needs to answer.

9 Acknowledgements

We thank our Shepherd Anil Somayaji and anonymous reviewers for the comments on the paper. This work was supported in part by the National Science Foundation the Cyber Trust program under Grant No. CNS-0716292.

References

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

[2] *NIST/SEMATECH e-Handbook of Statistical Methods.* http://www.itl.nist.gov/div898/handbook/, 2008.

[3] Athttpd Remote GET Request Buffer Overrun Vulnerability. http://www.securityfocus.com/bid/8709/discuss/, as of 2008.

[4] GNOME bug tracking system. http://bugzilla.gnome.org/, as of 2008.

[5] LIGHT http server and content management system. http://lhttpd.sourceforge.net/, as of 2008.

[6] miniLZO, a lightweight subset of the LZO library. http://www.oberhumer.com/opensource/lzo/#minilzo, as of 2008.

[7] Newspost, a usenet binary autoposter for unix. http://newspost.unixcab.org/, as of 2008.

[8] NullLogic, the Null HTTPD server. http://nullwebmail.sourceforge.net/httpd/, as of 2008.

[9] Privacy Statement for the Microsoft Error Reporting Service. http://oca.microsoft.com/en/dcp20.asp, as of 2008.

[10] Process Tracing Using Ptrace. http://linuxgazette.net/issue81/sandeep.html, as of 2008.

[11] Reducing Support Costs with Windows Vista. http://technet.microsoft.com/en-us/windowsvista/aa905076.aspx, as of 2008.

[12] Speed up Windows Mobile 5 pocket device . http://www.mobiletopsoft.com/board/388/speed-up-windows-mobile-5-pocket-device.html, as of 2008.

[13] Speed Up Windows Vista. http://www.extremetech.com/article2/0,1697,2110598,00.asp, as of 2008.

[14] Sumus Game Server Remote Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/13162, as of 2008.

[15] SUMUS, the mus server. http://sumus.sourceforge.net/, as of 2008.

[16] Technical Note TN2123, CrashReporter. http://developer.apple.com/technotes/tn2004/tn2123.html, as of 2008.

[17] Tip: Disable Error reporting in Windows Mobile 5 to get better performance: msg#00043. http://osdir.com/ml/handhelds.ipaq.ipaqworld/2006-05/msg00043.html, as of 2008.

[18] U.S. Department of Energy Computer Incident Advisory Capability. Office XP Error Reporting May Send Sensitive Documents to Microsoft. http://www.ciac.org/ciac/bulletins/m-005.shtml, as of 2008.

[19] VMPS, VLAN Management Policy Server. http://vmps.sourceforge.net/, as of 2008.

[20] Windows Error Reporting. http://msdn2.microsoft.com/en-us/library/bb513641%28VS.85%29.aspx, as of 2008.

[21] ABADI, M., BUDIU, M., ERLINGSSON, Ú., AND LIGATTI, J. Control-flow integrity. In *ACM Conference on Computer and Communications Security* (2005), pp. 340–353.

[22] BERKMAN, J. Project Info for Bug-Buddy. http://www.advogato.org/proj/bug-buddy/, as of 2008.

[23] BHANSALI, S., CHEN, W.-K., DE JONG, S., EDWARDS, A., MURRAY, R., DRINIĆ, M., MIHOČKA, D., AND CHAU, J. Framework for instruction-level tracing and analysis of program executions. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments* (2006), pp. 154–163.

[24] BRICKELL, J., PORTER, D. E., SHMATIKOV, V., AND WITCHEL, E. Privacy-preserving remote diagnostics. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security* (2007), pp. 498–507.

[25] BROADWELL, P., HARREN, M., AND SASTRY, N. Scrash: A system for generating secure crash information. In *Proceedings of the 12th USENIX Security Symposium* (Aug. 2003), USENIX, pp. 273–284.

[26] BRUMLEY, D., NEWSOME, J., SONG, D. X., WANG, H., AND JHA, S. Towards automatic generation of vulnerability-based signatures. In *S&P* (2006), pp. 2–16.

[27] BUGTOASTER. Do Something about computer Crashes. http://www.bugtoaster.com, 2002.

[28] CASTRO, M., COSTA, M., AND HARRIS, T. Securing software by enforcing data-flow integrity. In *OSDI* (2006), pp. 147–160.

[29] CASTRO, M., COSTA, M., AND MARTIN, J.-P. Better bug reporting with better privacy. In *Proceedings of Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)* (2008).

[30] COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A. I. T., ZHOU, L., ZHANG, L., AND BARHAM, P. T. Vigilante: end-to-end containment of internet worms. In *Proceedings of SOSP* (2005), pp. 133–147.

[31] CRANDALL, J. R., SU, Z., AND WU, S. F. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security* (New York, NY, USA, 2005), ACM Press, pp. 235–248.

[32] CUI, W., PAXSON, V., WEAVER, N., AND KATZ, R. H. Protocol-independent adaptive replay of application dialog. In *NDSS* (2006).

[33] DUTERTRE, B., AND MOURA, L. The YICES SMT Solver. http://yices.csl.sri.com/, as of 2008.

[34] EGELE, M., KRUEGEL, C., KIRDA, E., YIN, H., AND SONG, D. Dynamic spyware analysis. In *Proceedings of the 2007 USENIX Annual Technical Conference (Usenix'07)* (June 2007).

[35] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Automated whitebox fuzz testing. In *NDSS* (2008).

[36] HA, J., ROSSBACH, C. J., DAVIS, J. V., ROY, I., RAMADAN, H. E., PORTER, D. E., CHEN, D. L., AND WITCHEL, E. Improved error reporting for software that uses black-box components. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (2007), pp. 101–111.

[37] KING, J. C. Symbolic execution and program testing. *Commun. ACM 19*, 7 (1976), 385–394.

[38] LI, N., LI, T., AND VENKATASUBRAMANIAN, S. t-closeness: Privacy beyond k-anonymity and l-diversity. In *ICDE* (2007), IEEE, pp. 106–115.

[39] LIANG, Z., AND SEKAR, R. Fast and automated generation of attack signatures: a basis for building self-protecting servers. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security* (New York, NY, USA, 2005), ACM Press, pp. 213–222.

[40] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (2005), pp. 190–200.

[41] MACHANAVAJJHALA, A., KIFER, D., GEHRKE, J., AND VENKITASUBRAMANIAM, M. L-diversity: Privacy beyond k-anonymity. *TKDD 1*, 1 (2007).

[42] MOSER, A., KRÜGEL, C., AND KIRDA, E. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy* (2007), pp. 231–245.

[43] NEWSOME, J., BRUMLEY, D., FRANKLIN, J., AND SONG, D. X. Replayer: automatic protocol replay by binary analysis. In *ACM Conference on Computer and Communications Security* (2006), pp. 311–321.

[44] NEWSOME, J., AND SONG, D. X. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS* (2005).

[45] QIN, F., WANG, C., LI, Z., KIM, H.-S., ZHOU, Y., AND WU, Y. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO* (2006), pp. 135–148.

[46] SAMARATI, P., AND SWEENEY, L. Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression, 1998. Technical Report SRI-CSL-98-04.

[47] SHANNON, C., AND MOORE, D. The spread of the witty worm. *IEEE Security & Privacy 2*, 4 (July/August 2004), 46–50.

[48] SHANNON, C. E. A mathematical theory of communication. *Bell system technical journal 27* (1948).

[49] SIDIROGLOU, S., AND KEROMYTIS, A. D. Countering network worms through automatic patch generation. *IEEE Security and Privacy 3*, 6 (2005), 41–49.

[50] SINGH, S., ESTAN, C., VARGHESE, G., AND SAVAGE, S. Automated worm fingerprinting. In *Proceedings of OSDI* (2004), pp. 45–60.

[51] VACHHARAJANI, N., BRIDGES, M. J., CHANG, J., RANGAN, R., OTTONI, G., BLOME, J. A., REIS, G. A., VACHHARAJANI, M., AND AUGUST, D. I. RIFLE: An architectural framework for user-centric information-flow security. In *MICRO* (2004), IEEE Computer Society, pp. 243–254.

[52] WANG, X., LI, Z., XU, J., REITER, M. K., KIL, C., AND CHOI, J. Y. Packet vaccine: black-box exploit detection and signature generation. In *ACM Conference on Computer and Communications Security* (2006), pp. 37–46.

[53] YANG, J., SAR, C., TWOHEY, P., CADAR, C., AND ENGLER, D. Automatically generating malicious disks using symbolic execution. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy* (2006), pp. 243–257.