Better Bug Reporting With Better Privacy

Miguel Castro Manuel Costa Jean-Philippe Martin
Microsoft Research Cambridge
{mcastro,manuelc,jpmartin}@microsoft.com

Abstract

Software vendors collect bug reports from customers to improve the quality of their software. These reports should include the inputs that make the software fail, to enable vendors to reproduce the bug. However, vendors rarely include these inputs in reports because they may contain private user data. We describe a solution to this problem that provides software vendors with new input values that satisfy the conditions required to make the software follow the same execution path until it fails, but are otherwise unrelated with the original inputs. These new inputs allow vendors to reproduce the bug while revealing less private information than existing approaches. Additionally, we provide a mechanism to measure the amount of information revealed in an error report. This mechanism allows users to perform informed decisions on whether or not to submit reports. We implemented a prototype of our solution and evaluated it with real errors in real programs. The results show that we can produce error reports that allow software vendors to reproduce bugs while revealing almost no private information.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging; K.4.1 [*Computers and Society*]: Public Policy Issues–Privacy; D.4.6 [*Operating Systems*]: Security and Protection

General Terms Algorithms, Reliability, Security

Keywords Bug reports, Privacy, Symbolic execution, Constraint solving

1. Introduction

Software vendors collect error reports from users to improve the security and reliability of their software. This is important because it allows vendors to fix bugs in a timely manner, but it raises privacy concerns because error reports may include private user data or confidential company data. For example, Microsoft's error reporting technology [28] collects bug reports automatically from millions of Windows users and it is credited with a major improvement on software quality. Microsoft's privacy policies for this technology illustrate typical privacy concerns [26, 27]. The following paragraph from [26] is a good example:

> *"Reports might unintentionally contain personal information, but this information is not used to identify you or contact you. For example, a report that contains a snapshot of memory might include your name, part of a document you were working on, or data that you recently submitted to a website. If you are concerned that a report might contain personal or confidential information, you should not send the report."*

When generating error reports, there is a tradeoff between the software vendor's ability to reproduce the bug and loss of user privacy. To enable vendors to reproduce the bug, these reports should include the inputs that make the software fail, for example, the documents users were editing or the messages received by a server. However, vendors rarely include these inputs in reports because they may contain private user data. Instead, most error reports include dumps of small regions of memory, for example, the memory in the stacks of running threads. These dumps may be insufficient to reproduce the bug and they may still reveal private information (as discussed in the previous paragraph from [26]).

This paper proposes a solution to this problem. Our solution provides software vendors with error reports that contain input values to reproduce the bug while revealing less private information than existing approaches. Additionally, we provide a mechanism to measure the amount of information revealed in an error report. This mechanism allows users to perform informed decisions on whether or not to submit reports.

We generate error reports automatically when software fails by analyzing the failed execution. We use symbolic execution along the path followed by the failed execution to compute *path conditions*: any input that satisfies the path conditions causes the software to follow the same execution path until it fails. Then, we use a Satisfiability Modulo Theories (SMT) solver (e.g., [8, 16, 18]) to compute a different input that satisfies the path conditions, but is otherwise unrelated with the original input. The error report contains this new input and it identifies the failed program. Software vendors can use the new input to reproduce the bug. They can analyze the failed execution step-by-step in a debugger to reduce the time and cost to fix the bug. Yet the error report only reveals the information in the path conditions.

As illustrated by Microsoft's recommendation at the end of the previous paragraph from [26], it is important to help users decide whether or not they should send an error report. Currently, it is hard for users to perform informed decisions. Concerned users have to look at memory dumps to decide whether or not to send reports. We suspect that many users do not do this and, instead, conservatively decide not to submit any report.

We developed an application-independent technique that computes an upper bound on the number of bits about the original input that are revealed by an error report. We can measure the entropy of an error report by computing the set of byte strings with the same size as the original input that satisfy the path conditions. If a fraction α of the byte strings satisfy the conditions, the error report reveals $-\log_2(\alpha)$ bits about the original input. Since computing this value exactly is too expensive for large inputs, we compute an

ASPLOS'08, March 1–5, 2008, Seattle, Washington, USA.
Copyright © 2008 ACM 978-1-59593-958-6/08/03. . . $5.00

upper bound. We also compute an upper bound on the number of bits revealed about each individual byte of the original input. We provide these values to users to help them decide whether or not to send the error report.

We implemented a prototype of our techniques and evaluated them with five real errors in five real programs. The results show that we can generate error reports that allow software vendors to reproduce the bug while revealing very little information. For example, the report for our Microsoft Word error reveals only 0.2% of the bits in the original input document and we were unable to recover any text from the report using Microsoft's text recovery tools. Additionally, our error reports are small. For example, the report for Microsoft Word compresses to 5.1KB whereas the original input compresses to 926KB.

We believe that our techniques have other interesting applications. For example, we can use them to remove shell code from exploit input. Attackers can supply exploit input to vulnerable programs to gain control over their execution. We can use our techniques to generate new input that can be used to debug the vulnerability without the risk of executing attacker supplied code.

The rest of the paper is organized as follows. Section 2 presents an overview of our error report generation technique and discusses why it is needed. Sections 3 and 4 describe how we generate path conditions and Section 5 how we use them to generate a new input. Section 6 presents our technique to measure the information in an error report. Section 7 presents our results. Related work appears in Section 8 and we conclude in Section 9.

2. Overview and motivation

We use the example in Figure 1 to motivate the need for our technique and to illustrate how it works. The example is a simplified Web server with a buffer overflow error. Section 7.1 describes a similar error in the ghttpd Web server [1]. The function ProcessMessage is called immediately after the message msg is received from the network. If the message contains a GET request, the function copies the URL to the array url, obtains the name of the target host, and calls ProcessGet to handle the request. A message with a long URL can overflow url and corrupt the stack.

```
int ProcessMessage(int sock, char *msg) {
  char url[20];
  char host[20];
  int i=0;

  if (msg[0] != ’G’ || msg[1] != ’E’
      || msg[2] != ’T’ || msg[3] != ’ ’)
    return -1;

  msg = msg+4;
  while (*msg != ’\n’ && *msg != ’ ’) {
    url[i++] = *msg++;
  }
  url[i] = 0;

  GetHost(msg, host);
  return ProcessGet(sock, url, host);
}
```

Figure 1. Example code: simplified Web server with a buffer overflow error.

We compiled our example Web server with Microsoft Visual Studio 2005 with the option that inserts canaries to detect stack overflows [14]. Then we sent the HTTP GET request in Figure 2, which has a long URL, to our Web server. This caused the server to overflow the url array and to overwrite the return address of ProcessMessage on the stack. The compiler inserted checks detected the error when ProcessMessage was about to return.

As an example of the state of the art, we used the error reporting technology of Microsoft Windows XP to generate a report for this buffer overflow. Figure 3 shows part of the stack dump included in the generated report. The dump reveals all the information in the request message. Unfortunately, a request can encode private information, for example, the URL in this request associates a name with a product and a credit card number.

```
0.@. ...\.@..>@.p.@.......@.......@.......@./checkout
?product=embarassingProduct&name=Jo....p.....@.....w.
..(.......GET /checkout?product=embarassingnDoe&credi
tcardnumber=1122334455667788.122334455667788 HTTP/1.1
.Accept: */*.Accept-Language: en-gb.UA-CPU: x86.Accep
t-Encoding: gzip, deflate.User-Agent: Mozilla/4.0 (co
mpatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322
; .NET CLR 2.0.50727).Host: www.ecommercesite.com.Con
nection: Keep-Alive.*3@..$........@......72..82..$...
```

Figure 3. Part of the memory dump included in the report generated by Microsoft Windows XP for our example error.

We argue that no one wants this private information in the error report. The clients of the Web server do not want this information disclosed because it is embarassing and it reveals their credit card number. The company running the Web server may violate the privacy statement in their Web site by sending this bug report to the software vendor. The software vendor does not want to store bug reports with private data because it must put in place special procedures to prevent leaking the data. Additionally, the software vendor cannot share reports with third parties, for example, with software vendors that develop plug-ins for their Web server.

Our techniques can solve this problem. We generate an error report for this example with a new input that also overflows url but does not disclose any private information: it does not reveal any bits about any character in the original URL. Figure 4 shows the architecture of our error report generation system. Our error reports are generated in the background like the error reports generated by Windows XP.

To generate error reports, we must detect errors during production runs of the software without introducing high overhead. Our current prototype detects errors using Windows XP's error checking mechanisms, which include a combination of hardware, operating system, compiler inserted, and application specific error checking. In the example in Figure 1, we also detect the error using stack overflow checks inserted by the compiler.

We require a detailed instruction trace to compute path conditions but the instrumentation to collect this trace has high overhead. Therefore, we use low overhead input logging during production runs. When we detect an error, we use the log to replay execution with instrumentation to collect the trace. The log is truncated by taking a checkpoint and the checkpoint is treated as an additional input when computing path conditions. We can use existing techniques to log inputs and take checkpoints [19, 31, 38] with low overhead, but these are not yet implemented in the current prototype. This paper focuses on generating error reports given the log.

When an error is detected, we replay the log using dynamic binary instrumentation [5] to collect a trace of x86 instructions. We can work with unmodified x86 binaries. We may be unable to reproduce some non-deterministic errors during replay, for example, errors that depend on unlikely interleaving of instructions from different threads in a multiprocessor. But we can reproduce deterministic errors that only depend on the logged input and even non-deterministic errors that occur with high probability given the logged input. Additionaly, we can add more detailed instrumenta-

```
GET /checkout?product=embarassingProduct&name=JohnDoe&creditcardnumber=1122334455667788 HTTP/1.1
Accept: */*
Accept-Language: en-gb
UA-CPU: x86
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727)
Host: www.ecommercesite.com
Connection: Keep-Alive
```

Figure 2. Example input: contains private information and triggers the bug.

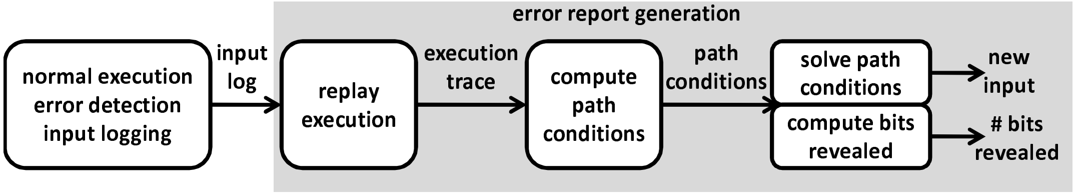

Figure 4. Error report generation architecture.

tion during replay to detect the error earlier, for example, we can add bounds checks [32] or data-flow integrity checks [9]. We have used data-flow integrity checks in some of the examples described in this paper.

We compute path conditions using the trace of x86 instructions executed from the point where input is first received until the point where the error is detected. These conditions are fed to an SMT solver [16] to compute a new input. The new input causes the software to follow the execution path in the replay but reveals nothing more about the original input. The software vendor can use the new input to reproduce the bug.

We can include either the path conditions or the new input in the error report. Both alternatives reveal the same amount of information about the original input. The first alternative saves computation time at the user machine but we chose the second because it makes it harder to launch denial-of-service attacks on the error reporting servers. It also saves bandwidth because the new input is smaller than the path conditions and it compresses better.

The path conditions are also used to compute the amount of information about the original input that is revealed by the report. We show the user the number of bits revealed by the report to help the user decide whether or not to submit the report.

3. Computing path conditions

Path conditions are conditions on the input of the faulty program such that replaying the execution with any input that satisfies them ensures that the program follows the same execution path until it fails (provided the program is deterministic). The execution path followed by non-deterministic programs may not be completely determined by the inputs we log. Therefore, the path conditions may not be sufficient to ensure the same execution path for non-deterministic programs, but they may still be sufficient with high probability. For example, the experimental results in [31] describe a data race error that can be reproduced with 60% probability.

We compute path conditions by performing forward symbolic execution along the trace collected during replay. The trace contains the sequence of x86 instructions executed by each thread and the concrete values of source and destination operands of each instruction. The symbolic execution starts by replacing the concrete values of the bytes in the logged input by symbolic values: the byte at index i gets symbolic value b_i. Then, it executes the instructions in the trace keeping track of the symbolic value of storage locations that are data dependent on the input. The symbolic values are expressions whose value depends on some of the b_i. They are represented as trees whose interior nodes are x86 instruction opcodes and whose leaves are constants or one of the b_i.

The symbolic execution defines a total order on the instructions in the trace that is a legal uniprocessor schedule. The instructions are processed one at a time in this total order. If the next instruction to be processed has at least one source operand that references a storage location with a symbolic value, the instruction is executed symbolically. Otherwise, any storage locations modified by the instruction are marked as concrete, that is, we delete any symbolic value these locations may have had because they are no longer data dependent on the input. For example, after executing `add eax,ebx` when `ebx` has symbolic value b_0 and `eax` has concrete value `10`, `eax` gets the symbolic value (`add` b_0 `10`).

Whenever the symbolic execution encounters a branch that depends on the input, it adds a new path condition to ensure that inputs that satisfy the path conditions can follow the execution path in the trace. A branch depends on the input if the value of `eflags` is symbolic. Path conditions are represented as a tree of the form: (`Jcc` f), where f is the symbolic value of `eflags`. If the branch is taken in the trace, `Jcc` is the opcode of the branch instruction. Otherwise, `Jcc` is the opcode of the branch instruction that tests the negation of the condition tested in the trace. Continuing the example above if `cmp eax,0;jg label` is executed in the trace and the branch is taken, symbolic execution generates the path condition (`jg` (`cmp` (`add` b_0 `10`) `0`)). If the branch had not been taken in the trace, the condition would be (`jle` (`cmp` (`add` b_0 `10`) `0`)). No conditions are added for branches that do not depend on the input.

Symbolic execution also generates path conditions when an indirect call or jump is executed and the value of the target operand is symbolic. The condition in this case asserts that $t_s = t_c$ where t_s is the symbolic value of the target and t_c is the concrete value of the target retrieved from the trace. We represent the condition as (`je` (`cmp` t_s t_c)). Similar conditions are generated when a load or store is executed and the address operand has a symbolic value. These conditions assert that $a_s = a_c$ where a_s is the symbolic value of the address operand and a_c is its concrete value retrieved from the trace. We represent the condition as (`je` (`cmp` a_s a_c)). EXE [8] describes a technique to generate weaker conditions in this case. We could use this technique to reveal less information but our current prototype only applies this technique to common library functions like `strtok` and `sscanf`.

```
00401037  mov       eax,dword ptr [msg]
0040103A  movsx     ecx,byte ptr [eax]
0040103D  cmp       ecx,47h
00401040  jne       ProcessMessage+46h (401066h)
00401042  mov       edx,dword ptr [msg]
00401045  movsx     eax,byte ptr [edx+1]
00401049  cmp       eax,45h
0040104C  jne       ProcessMessage+46h (401066h)
```

Figure 5. Fragment of the execution trace for our example.

We will use the execution trace obtained by sending the request in Figure 2 to the Web server in Figure 1 to illustrate how we compute path conditions. Figure 5 shows a fragment of this trace that corresponds to `msg[0] != 'G' || msg[1] != 'E'` in the source code. The first instruction loads the address of the message from the stack to `eax` and the second loads the first byte of the message (with sign extension) to `ecx`. Therefore, `ecx` has symbolic value (`movsx` b_0) at this point. The third instruction compares `ecx` with `0x47` ('G') and assigns the symbolic value (`cmp` (`movsx` b_0) `0x47`) to the zero flag. Since `jne` is not taken, the path condition for the fourth instruction is (`je` (`cmp` (`movsx` b_0) `0x47`)). The last four instructions in the figure are similar but they check the second byte. We generate the path condition (`je` (`cmp` (`movsx` b_1) `0x45`)) for the eight instruction. While processing the rest of the trace, we generate the additional conditions: (`je` (`cmp` (`movsx` b_2) `0x54`)), (`je` (`cmp` (`movsx` b_3) `0x20`)), and (`jne` (`cmp` (`movsx` b_i) `0xa`)) and (`jne` (`cmp` (`movsx` b_i) `0x20`)) for the remaining bytes in the input URL. There are also additional conditions from the execution of `GetHost` and `ProcessGet`.

4. Removing path conditions

The path conditions computed as described in the previous section are sufficient to generate new inputs that reproduce the bug, but some of them may not be necessary. By removing some of the unnecessary conditions, we can generate smaller bug reports that reveal less private information.

We use additional error checks and analysis during replay to remove unnecessary conditions without increasing the overhead during normal execution. The error report includes information about the type of analysis used during replay to allow the software vendor to reproduce the bug using the same analysis.

We used *data-flow integrity* analysis (DFI) [9] to remove unnecessary path conditions in some of the examples that we studied. We could also use, for example, bounds checking [32] and precondition slicing [12]. DFI adds checks to detect memory safety violations. Using static program analysis, DFI associates each instruction that reads data from memory with a set of instructions that are allowed to write the data. This defines the valid data-flows in a program. DFI then enforces these data-flows at runtime: whenever a value is read, DFI checks that the instruction that wrote the value is in the set of allowed writers. Thus, DFI detects a memory safety violation when an instruction reads data produced by an out-of-bounds write [9]. Usually, DFI checks can detect an error before the checks that we use during normal execution. Therefore, we can stop the replay earlier and generate less path conditions.

We also experimented with two improvements to remove additional path conditions. First, we traverse the trace backwards to find the instruction that wrote the data out-of-bounds and remove path conditions that were added after that point in the trace. Second, we analyze errors that corrupt the internal data structures in libraries where DFI does not check reads [9]. For example, many errors corrupt the heap management data structures in the C runtime, which can cause library code to write anywhere in memory. DFI detects the error when it reads data produced by this write. We implemented an analysis to find the instruction that first corrupts the heap management data structures. We first traverse the trace backwards to find the unsafe write. If this write was performed by one of the heap management functions (e.g., `malloc`), we traverse the trace forward from the begining to find the first read inside `malloc`, `calloc` or `free` of a value written outside these functions. We remove path conditions added after that point in the trace.

In our example from Figures 1 and 2, DFI detects the buffer overflow when the 21^{st} character in the URL is about to be written to `url`. At this point, the value of `i` has been overwritten by an out of bounds write to `url`. Therefore, when the program reads the value of `i`, DFI detects that the value was not produced by an instruction allowed to write to `i`. Since we stop replaying at this point, we can eliminate all the conditions added by `GetHost` and `ProcessGet` and only require conditions of the form (`jne` (`cmp` (`movsx` b_i) `0xa`)) and (`jne` (`cmp` (`movsx` b_i) `0x20`)) for $4 \leq i \leq 21$. None of the two improvements to DFI is useful in this case, but in section 7 we show that they can also remove path conditions.

5. Solving path conditions

We use a Satisfiability Modulo Theories (SMT) solver to compute a new input that satisfies the path conditions, but is otherwise unrelated with the original input. We include the new input in the error report to allow software vendors to reproduce the bug.

The current prototype uses `Z3 1.0` [16] to compute the new input but we could use other SMT solvers (e.g. [8, 17]). To use `Z3`, we convert the path conditions from the tree representation described in Section 3 to the bit vector types and primitives of the solver. For example, the path condition (`je` (`cmp` (`movsx` b_0) `0x47`)) is converted to (`=` (`sign_extend[24]` b_0) `bv71[32]`).

There are some x86 instructions that the current prototype cannot convert to the language of the solver, for example, it cannot convert floating point instructions. Any path condition C that involves these instructions is replaced by a set of conditions stating that the input bytes involved in computing C are equal to their concrete values in the original input. This ensures that the solution computed by the solver can be used to reproduce the bug but may reveal more information than necessary. Improving our prototype will reduce the amount of information revealed in error reports.

Even though it is not possible to give an upper bound on the time to compute a new input, modern SMT solvers are surprisingly fast. It takes `Z3` less than 14 seconds to compute a new input in all the examples that we looked at. Others report similar performance [8].

The new inputs generated by the solver compress very well. There are two reasons for this. It is common for many input bytes to have no constraints, i.e., not to appear in the path conditions. We assign the value zero to all these bytes to pad the new input to be the same size as the original input. Additionally, the solver assigns the same value to input bytes that have the same constraints in the path conditions, which is common because of loops. For example, the new input that we generate for the error in Microsoft Word XP (described in Section 7.3) compresses from 2.5MB to 5.1KB whereas the original input compresses to 926KB.

We used `Z3` to compute a new input satisfying the path conditions for the execution obtained by sending the request in Figure 2 to the Web server in Figure 1. The new input is shown in Figure 6 where '.' represents byte value zero. It is interesting to compare the new input with the original message in Figure 2. This new input reveals nothing about the URL in the original request except that it is longer than 21 bytes. Yet it is sufficient to reproduce the bug.

It is also interesting to compare the new input with the stack dump included in the error report generated by Windows XP for our example. As shown in Figure 3, the stack dump reveals all the

```
GET ....................
```

Figure 6. New input included in the error report for our Web server example. The '.' represents byte value zero.

information in the request message. Furthermore, stack dumps may not be sufficient to reproduce the bug.

6. Measuring privacy loss

We must ask the user for confirmation before sending an error report that may contain private information. To help the user make an informed decision, we developed a technique that computes an upper bound on the number of bits about the input that are revealed by an error report.

The reports that we generate reveal only the information in the path conditions: all inputs that satisfy the path conditions generate the same error report. So we can measure the entropy loss of an error report by computing the set of byte strings with the same size as the original input that satisfy the path conditions. If this set contains a single byte string, the report reveals all the information about the original input. If the set contains two byte strings of length l, the report reveals $l - 1$ bits. In general, if a fraction α of the byte strings satisfy the conditions, the error report reveals $-\log_2(\alpha)$ bits about the original input.

This is a pure entropy measure. It does not take into account any input structure that may be known beforehand, e.g., if some byte strings are more likely to occur than others. However, it has the advantage of being application independent. We can add application-specific knowledge to our prototype when it is available.

It is important to compute the entropy loss of an error report fast because the user needs this value to decide whether or not to send the report. Since computing α exactly is expensive for large inputs, we have developed an efficient mechanism to compute an upper bound on the number of bits that are revealed. We also compute an upper bound on the number of bits revealed about each individual byte of the original input to provide the user with additional information.

Our mechanism has two steps that we explain next. The first step computes the number of bits revealed by subsets of path conditions. The second step combines these partial results to get the final result.

6.1 Computing partial results

We start by parsing and simplifying the path conditions. Then we compute the number of bits revealed by conditions that reference a single input byte. For each input byte b_i, we compute the conjunction C_i of all the conditions that reference only b_i. Conjunction C_i reveals $-\log_2(\beta_i)$ bits about b_i, where β_i is the fraction of byte values that satisfy C_i. We iterate over all possible byte values to compute β_i. We perform a similar brute-force search for conditions that reference two input bytes.

This brute-force search is too slow for conditions that reference more than two input bytes. So we use an approximation to compute an upper bound on the number of bits revealed by each of these conditions. For each condition $(op\ f(.)\ g(.))$, we compute a *summary* for f and g and use a set of rules to compute the bound given the summaries. The summary of a function f contains:

- lower and upper bounds for the value of f (denoted $f.l$ and $f.h$)
- lower and upper bounds on the cardinality of f's range (denoted $f.lr$ and $f.hr$)
- the *homogeneous* flag ($f.hom$), which is true only if each image of f has the same number of preimages
- the *masked-homogeneous* flag ($f.mh$), which is true only if f is homogeneous and there exist v, w such that the range of f is exactly $\{x\&v|w$: for all $x\}$ (where & and | are bitwise conjunction and disjunction)
- the input bytes referenced by f and the bit width of the output.

For example, the function $f_1 \equiv$ (`bvand` b_i `1`) has the summary: $f_1.l = 0$, $f_1.h = 1$, $f_1.lr = f_1.hr = 2$, and it is both homogeneous (each image has 128 preimages) and masked-homogeneous ($v = 1, w = 0$). The function $f_2 \equiv$ (`bvadd` (`bvand` b_i `1`) (`bvand` b_i `3`)) has three images: 0, 2, and 4. Its summary is: $f_2.l = 0$, $f_2.h = 4$, $f_2.lr = f_2.hr = 3$ and it is neither homogeneous nor masked-homogeneous because 2 has 128 preimages and 0 and 4 have 64 each.

Additionally, we use the summary of a function to compute its *lower density*, that is, the minimal number of preimages that any image of the function can have. We denote the lower density of a function f by $f.ld$. If f is homogeneous, $f.ld =$ `input-count`$(f)/f.hr$, where `input-count`(f) is the cardinality of f's domain. For example, f_1 has lower density $f_1.ld = 128$. If f is not homogeneous, it is hard to compute its lower density. So we conservatively assume that it is one. For example, we would assume $f_2.ld = 1$ even though the actual lower density for f_2 is 64. To get tight bounds, it is important to track which functions are homogeneous accurately. The masked homogeneous property allows us to identify more cases where a function is homogeneous in the presence of bitwise operations. For example, knowing that f is homogeneous does not guarantee that $f|1$ is homogeneous. But if f is masked-homogeneous, $f|1$ is masked-homogeneous and, therefore, it is also homogeneous. In some rules, we also compute the *higher density* of a function, which is the maximal number of preimages that any image of the function can have.

```
double UnsignedLessThan(Summary f, Summary g) {
  if (f.h < g.l) { return 0 }
  wcRange := g.l - (f.h+1 - f.lr)
  if (wcRange >= 1) {
    accepted := min(wcRange*f.ld, input-count(f))
  } else if (f,g have input byte in common) {
    accepted := 1
  } else { accepted := f.ld }
  return -log2( accepted / input-count(f)} )
}
```

Figure 7. Number of bits revealed by "$f < g$"

Figure 7 shows the rule to compute an upper bound on the number of bits revealed by an unsigned "<" condition given the summaries of its operands f and g. If the upper bound of f is less than the lower bound of g, we reveal zero bits because the condition holds for any value of the input. Otherwise, we compute a lower bound `wcRange` on the number of images of f that fall below $g.l$. This value is minimized when f has only $f.lr$ images and all these images are near $f.h$. If `wcRange` is at least one, we obtain a lower bound on the number of accepted inputs by multiplying `wcRange` by f's lower density. Otherwise, there must be at least one input x that is accepted because the condition is satisfiable. Additionally, if f and g have no input bytes in common, at least $f.ld$ inputs are accepted because all the inputs with the same image as x are also accepted. The last line returns the upper bound on the number of bits revealed. We have similar rules for other operators [24].

We also have rules to compute the summary of an expression given the summaries of its operands. We compute the summary of each operand of a path condition by applying these rules bottom up to its expression tree. Figure 8 shows a rule that computes a summary m for a multiplication given summaries f and g for the operands. The first check determines whether the multiplication can overflow (`bitwidth` is the number of bits in the multiplication result). If it cannot, the lower bound $m.l$ and the upper bound $m.h$

```
Summary multiply(Summary f, Summary g) {
  Summary m;
  if (f.h * g.h < 2^bitwidth) {
     m.l := f.l * g.l;
     m.h := f.h * g.h;
     if ((f, g have input byte in common)
         or (f.lr==1 && f.l==0)
         or (g.lr==1 && g.l==0))
     then m.lr := 1;
     else m.lr := max(f.lr, g.lr);
     m.hr := min(f.hr * g.hr, 2^bitwidth);
     m.hom := (is-constant(g) && f.hom);
     m.mh := (f.mh && is-constant(g) && g.l is power of 2);
     return m;
  }
  m.l := 0; m.h := 2^bitwidth-1; m.lr := 1;
  m.hr := 2^bitwidth; m.hom := false; m.mh := false;
  return m;
}
```

Figure 8. Summary rule for "$f * g$"

are obtained by multiplying the corresponding bounds for f and g. Then we compute a lower bound on the range. There are two cases where the bound may drop to one: f and g reference an input byte in common (e.g., $f \equiv$ (bvand b_i 1) and $g \equiv$ (bvand (bvneg b_i) 1)), or one of the functions could be the constant zero. In either case, we set $m.lr$ to one. Otherwise, $m.lr$ is the maximum of the corresponding bounds for f and g because $f * g$ has at least as many images as both f and g. Since there is no overflow, $m.hr$ is the product of $f.hr$ and $g.hr$. We set the homogeneous flag to false unless f is homogenous and g is a constant. The masked-homogeneous property is preserved if f is multiplied by a power of two because $(x\&v|w) * 2^c = (x * 2^c)\&(v * 2^c)|(w * 2^c)$. Otherwise, $m.mh$ is set to false. In the case of overflow, we compute the summary conservatively. We have similar rules for other operations [24].

6.2 Combining partial results

We must combine the partial results obtained in the previous step to compute upper bounds on the number of bits revealed by the error report for each input byte and for the whole input. We combine two groups of conditions by taking their conjunction. If the two groups have no input bytes in common, the number of bits revealed by their conjunction is the sum of the number of bits each reveals. Otherwise, we compute the number of inputs accepted by each group (*accepted*$_1$ and *accepted*$_2$), and we use these values to compute a lower bound *accepted*$_{1\wedge 2}$ on the intersection of the sets of inputs accepted by each group:

accepted$_{1\wedge 2}$=*max*(1, *accepted*$_1$ + *accepted*$_2$ - *inputCount*).

where *inputCount* is the number of byte strings with length equal to the number of distinct input bytes in the conjunction. Then we compute an upper bound on the number of bits revealed by the conjunction using the formula $-\log_2($*accepted*$_{1\wedge 2}$/*inputCount*$)$.

To compute the upper bound on the number of bits that are revealed about the whole input, we create an undirected graph with a node for each condition or conjunction of a group of conditions, and an edge between nodes that have an input byte in common. Then we compute the connected components of this graph and combine all the nodes in each component as described above. Once each component is reduced to a single combined condition, we sum the number of bits revealed by the components to determine the number of bits revealed about the whole input. To compute the upper bound on the number of bits that are revealed about each individual byte, we combine the groups of conditions that refer to that input byte as described above.

We used our algorithm to compute upper bounds on the number of bits revealed by our example error report (in Figure 6) for each input byte and for the whole input. We implemented a tool that prints a *leak graph* showing the upper bound on the number of bits revealed for each individual byte in the original input. The leak graph for our example error report is:

```
GET .....................
```

where the first four bytes are entirely revealed and we reveal between 0 and 1 bit for the next 21. The tool reports that the total number of bits revealed for the whole input is 32.2, which is accurate in this case. We reveal eight bits for the first four bytes and we reveal $-\log_2(254/256)$ bits for the next 21 bytes because all byte values but two satisfy the conditions on each of those input bytes: $4 * 8 - \log_2(254/256) * 21 = 32.2$.

7. Evaluation

We implemented a prototype of our error reporting system and we evaluated the system by generating bug reports for real crashes of five real programs: Ghttpd, Microsoft SQL server, Nullhttpd, Libpng, and Microsoft Word. We measured the amount of information revealed by each report. The results show that we can generate bug reports that allow vendors to reproduce the bugs easily while revealing almost no private information. We also measured the report sizes: all the reports had less than 5.1KB. Finally, we measured the time to generate each report and the contribution of each phase to the total time. The results show that all the reports were generated in less than 100 seconds, which is unlikely to inconvenience users because the reports are generated in the background.

All the experiments ran on a Dell Latitude D620 computer with one 2.16 GHz Intel Core2 Duo processor and 2GB of memory. We used the Windows Vista operating system and version 14 of Microsoft's C++ compiler. We used version 1.0 of the Z3 SMT solver [16].

7.1 Programs and crashes

We start by describing the programs that we studied and the crashes that we observed. It is hard to find a detailed description of a bug together with inputs that reproduce the bug. The exception are software vulnerabilities and inputs that exploit them, which are readily available on the Internet. Therefore, most of the errors that we studied are software vulnerabilities and we modified published exploits to crash the programs.

Ghttpd is an HTTP server with several vulnerabilities [1]. We crashed Ghttpd using a stack buffer overflow when processing the target URL for GET requests. The overflow occurs when logging the request inside a call to vsprintf. We caused the overflow by typing a URL with 392 characters in Internet Explorer 7 (IE7). The Web browser formats a message starting with GET and followed by the URL and a few more HTTP protocol elements. Figure 10 shows the message sent to Ghttpd by IE7. Ghttpd crashes when receiving this message. Any user submiting a URL with more than 150 bytes to the server would cause a similar crash.

Microsoft SQL server 2000 is a relational database that was infected by the infamous Slammer [30] worm. We crashed SQL Server using the buffer overflow vulnerability exploited by Slammer: we sent a UDP message to port 1434 with the first byte equal to 0x4 followed by 75 'A' characters. The characters following the first byte are the name of a database instance. While copying these bytes inside a call to sprintf, SQL Server overflows a buffer on the stack and crashes. Any user of the management tools looking up a database instance with a name longer than 60 characters would cause a similar crash.

Nullhttpd is another HTTP server. This server has a heap overflow vulnerability that an attacker can exploit by sending HTTP POST requests with a negative content length field in the

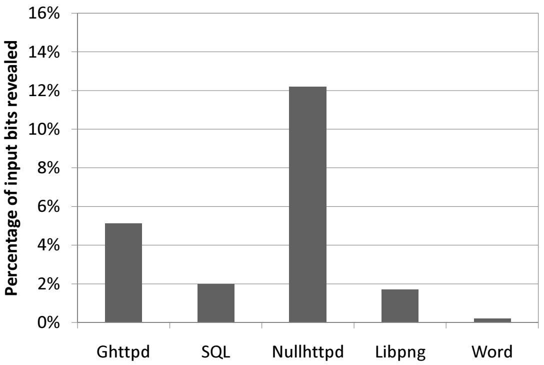

Figure 9. Percentage of input bits revealed in bug reports.

header [2]. These requests cause the server to allocate a heap buffer that is too small to hold the data in the `POST` request. While calling `recv` to read the `POST` data into the buffer, the server overwrites the heap management data structures maintained by the C library. We crashed `Nullhttpd` with the exploit described in [10]. This is a two message exploit. The first message exploits the vulnerability to modify the CGI-BIN configuration string to allow the attacker to start an arbitrary program. This first message is shown in Figure 12. It starts with `POST` and a URL, followed by `HTTP/1.0` and a `Content-Length` field of `-800`, followed by a line with a cookie, and followed by the `POST` data. The second message, which normally starts a shell, crashes the server, because we compiled `Nullhttpd` with data-flow integrity checks [9] (as described in Section 4).

`Libpng` is a library for processing images in the PNG file format [3]. Many applications, including MSN Messenger [25], use `Libpng` to display images. We built a test application distributed with `Libpng` and crashed it using the vulnerability described in [29]. The PNG format is based on chunks. We created an image file with a colour type of 0x03 in the IHDR chunk, with a tRNS chunk with more than 256 bytes, and without a PLTE chunk. This file causes the application to overflow a stack buffer and crash. Similar images could be generated by buggy image manipulation programs and they would trigger the same crash.

`Microsoft Word XP` is a widely used word processing application. We crashed `Word` by opening a document that contains 2.48 MB of mostly text. This is a confidential document that contains the specification of a real product. As far as we could determine, this bug cannot be exploited to gain control of `Word`. Since this is not a security vulnerability, there is little information available about the bug that causes the crash. Our system does not require any such information. It only uses the unmodified `Word` binary and the input document that causes the crash to compute the error report.

For each of our test cases, we replayed the error, gathered an execution trace, computed path conditions, and generated new inputs to include in the bugs reports.

7.2 Information revealed

We used the algorithm described in Section 6 to compute an upper bound on the number of bits about the original input revealed by each of the bug reports. Figure 9 shows the results. For `SQL Server`, `Libpng`, and `Word`, the bug reports reveal less than 2% of the bits in the original input. The reports for `Nullhttpd` and `Ghttpd` reveal 12.2% and 5% of the input bits, respectively. A detailed inspection of the code and the path conditions shows that we do not reveal any bits about user supplied data.

The report for `SQL Server` contains a message with the first byte equal to 0x4 followed by 75 characters different from 0x0.

```
POST / HTTP/1.0
Content-Length: -800
Cookie: TTTTTTTTTT

NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
```

Figure 12. Message that causes Nullhttpd to crash. `POST` data is truncated.

```
POST . .....1.0
Content-Length: -800
Cookie: ..........

............................................
............................................
```

Figure 13. Leak graph for the Nullhttpd bug report. Bytes marked with '.' leak 0 to 1 bits; other bytes leak at least 5 bits.

The report reveals no information about the name of the database instance that caused the crash, except its size.

The report for `Libpng` contains a PNG image with a colour type of 0x03 in the IHDR chunk and with a tRNS chunk with more than 256 bytes. Since all the bytes in the image chunk are set to zero, the report does not reveal anything about the image except its size.

We only reveal 0.19% of the bits in the original `Word` document. It is harder to analyze the contents of the report for `Word` in detail because the `Word` file format is complex. We used `Word`'s "recover text from any file" tool on both the original document and the document that we generated. The tool recovers 28,295 words from the original document. These words include a large amount of text and the names of the authors and their affiliation. Even though the document that we generated triggers the same error, the tool only recovers the following words from our document: `WordDocument`, `1Table`, and `©`. These words are associated with formatting elements and reveal no private user data.

We also computed an upper bound on the number of bits revealed about each individual byte of the original input. We show this information in the form of *leak graphs* for `Ghttpd` and `Nullhttpd` in Figures 11 and 13, respectively. For each byte in the original input, a leak graph contains the original byte value if the report reveals at least 5 bits about that byte or '.' when the report reveals 0 to 1 bits about the original byte. We generated these leak graphs using the algorithm in Section 6 but we corrected imprecisions of our approximation by carefully examining the path conditions. The leak graphs generated by our algorithm were very precise. They only differed from the ones we show on two byte positions for each crash, for example, our algorithm conservatively estimated that the report for `Nullhttpd` revealed the last two bytes in the cookie, which is not the case.

The graphs for `Ghttpd` and `Nullhttpd` can be compared with the corresponding original inputs in Figures 10 and 12. The graph for `Ghttpd` shows that the byte strings "`GET `", and "`HOST:`" are leaked in the report, as well as the first character of "`HTTP`". So we only reveal protocol bytes. We do not reveal anything about the URL or the virtual host name except their sizes.

The leak graph for `Nullhttpd` shows that the strings "`POST `", "`Content-Length: `", and "`Cookie: `" are leaked in the report. These are protocol tokens that contain no private information. The report also leaks that the `Content-Length:` is "-800". The values of the `POST` data, URL, and `Cookie` are not revealed.

It is harder to read leak graphs for binary formats like those of `Word` and `libpng`. Software vendors could provide application-specific tools to visualize leak graphs, for example, we effectively used `Word`'s "recover text from any file" to visualize a leak graph.

```
GET /jjhjkdshjkdshkjdshkjdhsjkdhskjdhsjkdshjdhsdsjkhdsjkhdskjhdskjhdskjdhsjkdshjkhkhjdskjhdskjhdskjhdskjhdskjdshkjdshkj
dhskjhjdskdjdjdsjhdskjhdsjkhdskjhdskjhdskjdhskjdshjkdjdsdjdskjhdskjhdskjhdskjdshkjdhskjhjdsdkjdskjhdskjhdskjdhkjddjdskj
dhskjdhskjhkjhdskjdhskjdkjdskjhdskjdshkdjjdskjdhskjdhskdjskjhdskjhdskjdhjdkjhdskjhdskjdhsjdskjhdskjdhskjhdskjdhskjdhsdk
jhdskjhdskjdhdhskjhkjhdskdhdskjdhskjddsj HTTP/1.1
Accept: */*
Accept-Language: en-gb
UA-CPU: x86
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727)
Host: 127.0.0.1
Connection: Keep-Alive
```

Figure 10. Message that causes Ghttpd to crash.

```
GET .................................................................................................................
.....................................................................................................................
.....................................................................................................................
................................... H.......
...........
.....................
...........
.............................
..................................................................................................
Host:..........
.....................
```

Figure 11. Leak graph for the Ghttpd bug report. Bytes marked with '.' leak 0 to 1 bits; other bytes leak at least 5 bits.

Finally, it is interesting to note that if we had embedded shellcode in the inputs for Ghttpd, SQL Server, Nullhttpd and Libpng, the new inputs would include none of the shellcode. This would enable humans to analyze the bug without risk of executing attack code. We could also use the new inputs to generate *self-certifying alerts* without shellcode [13].

7.3 Report size

After generating new inputs, we compress them before sending the bug report over the network. Figure 14 shows sizes of the compressed inputs included in each bug report. For SQL Server, Nullhttpd, and Libpng the bug reports have less than 200 bytes. The bug reports for Ghttpd and Word have 0.5KB and 5.1KB, respectively. All the reports are small, hence they can be sent over the network efficiently.

The sizes of the reports for Libpng and Word are interesting, because in both cases the input that causes the crash is substantially larger: 2.5MB for Word and 3KB for Libpng. Even if we compress the original PNG and Word files, the size of the compressed inputs is much larger than the size of the inputs included in the reports. Figure 15 illustrates this by showing the size of the bug reports normalized by the size of the compressed original inputs (the compressed original inputs are 926KB for Word and 2.4KB for Libpng). The figure shows that the PNG image and Word document in the reports are an order of magnitude more compressible than the original documents. This is expected because all the image bytes in the PNG file and many of the bytes in the Word document are set to 0x0 because they are not referenced in the path conditions. Furthermore, all the bytes that are constrained by identical conditions will have identical values in the bug report.

For SQL Server, this effect is not noticeable because the original input is very small. For both Ghttpd and Nullhttpd the original inputs are compressible by a factor of 10, again making this effect less important. The report for Ghttpd is slightly larger than the compressed original input, because some identical input bytes are processed by different execution paths, leading to different constraints and different bytes in the report.

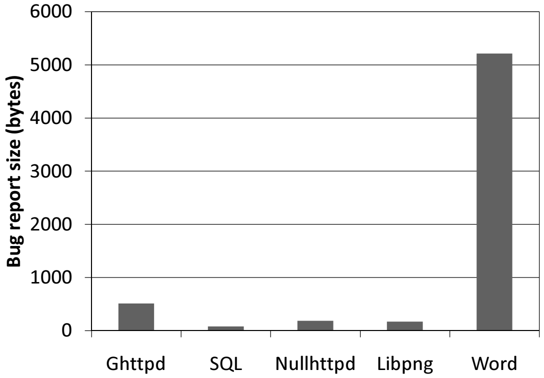

Figure 14. Size of bug reports.

7.4 Generation time

The last experiment measured the total time it takes to generate each of the bug reports. Figure 16 shows the results. The reports for Ghttpd, SQL Server, Nullhttpd and Libpng are generated in 10 seconds or less. The report for Word is generated in 90 seconds. These times are sufficiently short not to inconvenience users because the reports are generated in the background.

Figure 17 shows how much tracing and symbolic execution to generate path conditions, constraint solving, and estimation of privacy loss (i.e. information leakage) contribute to the total report generation time. In all cases, except Ghttpd, symbolic execution is the largest contributor to the generation time. For Ghttpd, 70% of the generation time is spent in the constraint solver.

8. Related work

Microsoft's error reporting technology [28] represents the state of the art in bug reporting. It collects bug reports automatically from millions of Windows users. These reports are used not only

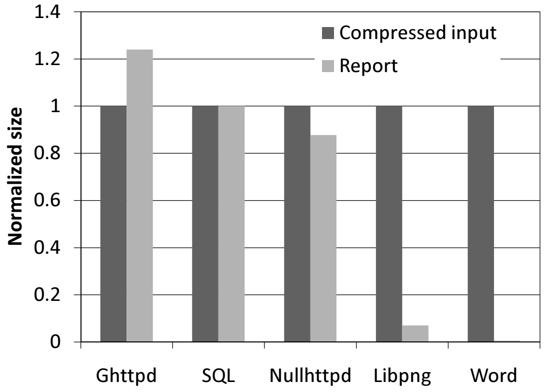

Figure 15. Size of bug reports normalized by size of compressed original input.

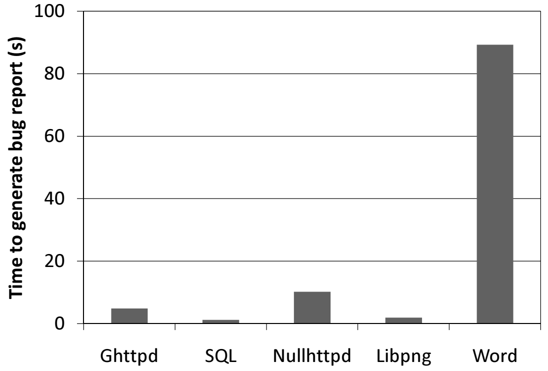

Figure 16. Time to generate bug reports.

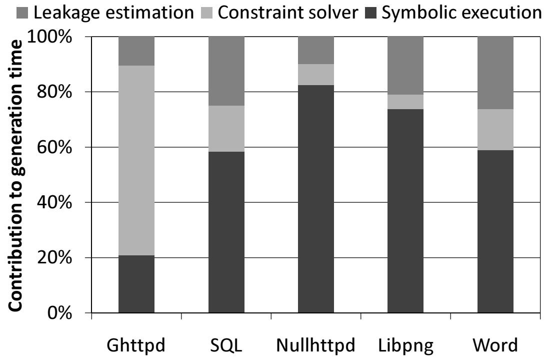

Figure 17. Contributions to report generation time.

to identify bugs but also to prioritize bug fixing. They include a description of the running program, a dump of the CPU state, and a dump of the stacks of running threads. They usually do not include the inputs necessary to reproduce the bug. In some rare cases, Microsoft may request additional information to fix an error, for example, input documents [27]. Microsoft's privacy policies for error reporting illustrate typical privacy concerns [26, 27]. We generate better error reports that reveal less private information than the state of the art.

Previous work has used similar techniques to compute path conditions but has applied these techniques to different problems. For example, Vigilante [13], DACODA [15], and Brumley et al [7] generate path conditions from execution traces collected during attacks. They use these conditions to filter out attack messages that exploit the same vulnerability.

DART [20], CUTE [35], and EXE [8] instrument C programs to collect path conditions and use an SMT solver to generate test inputs automatically. DART starts by computing path conditions for an execution with a random input. Then it takes a prefix of the conditions, negates the last one, and feeds the resulting conditions to an SMT solver to obtain a new test input that explores a different path. Sage [21] is similar to DART but works with x86 binaries. We solve a different problem: we improve the collection of test cases from a potentially very large population of users.

Our measure of privacy loss estimates the entropy loss [36] of the information in the error reports assuming no a-priori knowledge about the relative likelihoood of input byte strings. If for a given application the a-priori likelihood of each byte string were known, we could take it into account. For example, a report revealing that the original input is one of a very rare set of inputs reveals more information than if the input were one of a set of the same size containing common inputs. Chirayath et al. [11] start from this concept when they define privacy loss, which is a measure of the amount of information about a given database that is released when certain statistical properties of the database (e.g. the mean value) are revealed. Our privacy loss estimate has the advantage of being application independent.

Another way of quantifying privacy loss is the k-anonymity measure [33, 37], where a data release is said to be k-anonymous if "the information for each person contained in the release cannot be distinguished from at least $k-1$ individuals whose information also appears in the release" [37]. k-anonymity is particularly attractive for databases that contain personally identifiable information but it cannot be applied in our domain.

The algorithm we use to estimate privacy loss has been designed to compute an upper bound quickly. One could also transform the path conditions into the equivalent boolean satisfiability problem and use a #SAT solver (#SAT solvers find the number of assignments to variables that satisfy the given conditions. See e.g. [22, 23, 34]). In the examples we investigated, our fast upper bound was already very close to the optimal answer (as shown by the leak graphs).

Some related work requires the private information to be marked beforehand. In Scrash [6], variables that represent information deemed private must be annotated by the programmer, and the compiler determines the set of variables that may hold their values. The content of these variables is then scrubbed from core dumps. Our approach is different: Scrash returns core dumps rather than bug-reproducing inputs, and we do not rely on a programmer to add the right annotations.

Zeller and Hildebrandt [39] describe a system that searches for a shorter input that still makes the program fail. The system stops when it finds an input such that removing any single byte would allow the program to succeed. Their approach requires running the program to failure repeatedly (instead of once in our case). It may stop at a local minimum and it does not guarantee that the new input triggers the same bug as the original input. Additionally, it reveals all the bits in the bytes that remain in the generated input.

There is a large body of work (see e.g. [4]) that is concerned with how to modify a database before release so as to maintain the statistical properties of the database (e.g. the distribution of salaries) but prevent data about specific people from being retrieved (e.g. the salary of an individual). A common technique to implement this privacy-preserving data mining is to add random noise to the data but this is not useful in our case.

9. Conclusions

Automatic collection of bug reports is important to help software vendors improve the quality of their products. Current bug reports consist mostly of snapshots of small regions of memory. They may contain private user data and they may be insufficient to reproduce the bugs. We address both of these problems by generating bug reports that include inputs that make the software fail, making it easy to reproduce the bugs, while leaking less private information than existing approaches.

We produce these bug reports by generating new inputs that make the software follow the execution path that fails but are otherwise unrelated with the inputs that caused the crash originally. Moreover, we compute an upper bound on the amount of private information leaked by a bug report and the input bytes that are revealed, allowing users to make informed decisions on whether to submit the report or not.

We evaluated the system by generating bug reports for real crashes of real applications. Our results show that the reports contain almost no private information and allow the crashes to be reproduced easily. Additionally, the reports are small and we can generate them quickly.

References

[1] GHttpd Log() Function Buffer Overflow Vulnerability (Bugtraq ID: 5960). http://www.securityfocus.com/bid/5960.

[2] Null HTTPd Remote Heap Overflow Vulnerability (Bugtraq ID: 5774). http://www.securityfocus.com/bid/5774.

[3] Portable network graphics (png) specification and extensions. http://www.libpng.org/pub/png/spec/.

[4] AGRAWAL, R., AND SRIKANT, R. Privacy-preserving data mining. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data* (2000), pp. 439–450.

[5] BHANSALI, S., CHEN, W.-K., DE JONG, S., EDWARDS, A., MURRAY, R., DRINIC, M., MIHOCKA, D., AND CHAU, J. Framework for instruction-level tracing and analysis of program executuions. In *VEE* (June 2006).

[6] BROADWELL, P., HARREN, M., AND SASTRY, N. Scrash: a system for generating secure crash information.

[7] BRUMLEY, D., NEWSOME, J., SONG, D., WANG, H., AND JHA, S. Towards automatic generation of vulnerability signatures. In *IEEE Symposium on Security and Privacy* (May 2006).

[8] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. EXE: Automatically Generating Inputs of Death. In *13th ACM Conference on Computer and Communications Security* (2006).

[9] CASTRO, M., COSTA, M., AND HARRIS, T. Securing software by enforcing data-flow integrity. In *OSDI* (Nov. 2006).

[10] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-control-data attacks are realistic threats. In *USENIX Security Symposium* (July 2005).

[11] CHIRAYATH, V., LONGPRE, L., AND KREINOVICH, V. Measuring privacy loss in statistical databases. In *Workshop on Descriptional Complexity of Formal Systems* (June 2006), pp. 16–25.

[12] COSTA, M., CASTRO, M., ZHOU, L., ZHANG, L., AND PEINADO, M. Bouncer: Securing Software by Blocking Bad Input. In *SOSP* (Oct. 2007).

[13] COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L., AND BARHAM, P. Vigilante: End-to-End Containment of Internet Worms. In *SOSP* (Oct. 2005).

[14] COWAN, C., PU, C., MAIER, D., HINTON, H., WADPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. Stackguard: Automatic detection and prevention of buffer-overrun attacks. In *USENIX Security Symposium* (Jan. 1998).

[15] CRANDALL, J. R., SU, Z., WU, S. F., AND CHONG, F. T. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *ACM CCS* (Nov. 2005).

[16] DE MOURA, L., AND BJORNER, N. Z3: An Efficient SMT Solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (Apr. 2008).

[17] DUTERTRE, B., AND DE MOURA, L. The YICES SMT Solver. http://yices.csl.sri.com.

[18] DUTERTRE, B., AND DE MOURA, L. A fast linear-arithemic solver for dpll(t). In *CAV06* (Aug. 2006).

[19] ELNOZAHY, E. N., ALVISI, L., WANG, Y.-M., AND JOHNSON, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys 34*, 3 (Sept. 2002), 375–408.

[20] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: Directed Automated Random Testing. In *PLDI* (2005).

[21] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Automated whitebox fuzz testing. Tech. Rep. MSR-TR-2007-58, Microsoft Research Technical Report, May 2007.

[22] GOMES, C. P., HOFFMANN, J., SABHARWAL, A., AND SELMAN, B. From sampling to model counting. In *IJCAI* (2007), pp. 2293–2299.

[23] GOMES, C. P., SABHARWAL, A., AND SELMAN, B. Model counting: A new strategy for obtaining good bounds. In *AAAI* (2006).

[24] MARTIN, J.-P. Upper and lower bounds on the number of solutions. Tech. Rep. MSR-TR-2007-164, Dec. 2007.

[25] MICROSOFT CORPORATION. Msn messenger. http://messenger.msn.com.

[26] MICROSOFT CORPORATION. Privacy statement for the microsoft error reporting service, Oct. 2005. http://oca.microsoft.com/en/dcp20.asp.

[27] MICROSOFT CORPORATION. Description of the end user privacy policy in application error reporting when you are using office. Microsoft Knowledge Base Q283768, Jan. 2007. http://support.microsoft.com/kb/283768.

[28] MICROSOFT CORPORATION. Dr. watson overview, Jan. 2007. http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/drwatson_overview.mspx?mfr=true.

[29] MITRE CORPORATION. Multiple buffer overflows in libpng 1.2.5. CVE-2004-0597, June 2004. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2004-0597.

[30] MOORE, D., PAXSON, V., SAVAGE, S., SHANNON, C., STANIFORD, S., AND WEAVER, N. Inside the Slammer worm. *IEEE Security and Privacy 1*, 4 (July 2003).

[31] QIN, F., TUCEK, J., SUNDARESAN, J., AND ZHOU, Y. Rx: Treating bugs as allergies - a safe method to survive software failures. In *SOSP* (Nov. 2005).

[32] RUWASE, O., AND LAM, M. A practical dynamic buffer overflow detector. In *NDSS* (Feb. 2004).

[33] SAMARATI, P., AND SWEENEY, L. Generalizing data to provide anonymity when disclosing information. In *Proceedings of the 17th Symposium on Principles of Database Systems* (1998), p. 188.

[34] SANG, T., BEAME, P., AND KAUTZ, H. A. Heuristics for fast exact model counting. In *SAT* (2005), pp. 226–240.

[35] SEN, K., MARINOV, D., AND AGHA, G. CUTE: A Concolic Unit Testing Engine for C. In *ESEC/FSE* (2005).

[36] SHANNON, C. E. A mathematical theory of communication. *SIGMOBILE Mob. Comput. Commun. Rev. 5*, 1 (2001), 3–55.

[37] SWEENEY, L. k-anonymity: a model for protecting privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst. 10*, 5 (2002), 557–570.

[38] TUCEK, J., LU, S., HUANG, C., XANTHOS, S., AND ZHOU, Y. Triage: diagnosing production run failures at the user's site. In *SOSP* (Nov. 2007).

[39] ZELLER, A., AND HILDEBRANDT, R. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng. 28*, 2 (2002), 183–200.