Camouflage: Automated Sanitization of Field Data

James Clause
College of Computing
Georgia Institute of Technology
clause@cc.gatech.edu

Alessandro Orso
College of Computing
Georgia Institute of Technology
orso@cc.gatech.edu

ABSTRACT

Privacy and security concerns have adversely affected the usefulness of many types of techniques that leverage information gathered from deployed applications. To address this issue, we present a new approach for automatically sanitizing failure-inducing inputs. Given an input I that causes a failure f, our technique can generate a sanitized input I' that is different from I but still causes f. I' can then be sent to the developers to help them debug f, without revealing the possibly sensitive information contained in I. We implemented our approach in a prototype tool, CAMOUFLAGE, and performed an empirical evaluation. In the evaluation, we applied CAMOUFLAGE to a large set of failure-inducing inputs for several real applications. The results of the evaluation are promising; they show that CAMOUFLAGE is both practical and effective at generating sanitized inputs. In particular, for the inputs that we considered, I and I' shared no sensitive information.

1. INTRODUCTION

Investigating techniques that capture data from deployed applications to support in-house software engineering tasks is an increasingly active and successful area of research (*e.g.,* [1, 3–5, 13, 14, 17, 21, 22, 26, 27, 29]). However, privacy and security concerns have prevented widespread adoption of many of these techniques and, because they rely on user participation, have ultimately limited their usefulness. Many of the earlier proposed techniques attempt to sidestep these concerns by collecting only limited amounts of information (*e.g.,* stack traces and register dumps [1, 3, 5] or sampled branch profiles [26, 27]) and providing a privacy policy that specifies how the information will be used (*e.g.,* [2, 8]). Because the types of information collected by these techniques are unlikely to be sensitive, users are more willing to trust developers. Moreover, because only a small amount of information is collected, it is feasible for users to manually inspect and sanitize such information before it is sent to developers.

Unfortunately, recent research has shown that the effectiveness of these techniques increases when they can leverage large amounts of detailed information (*e.g.,* complete execution recordings [4, 14] or path profiles [13, 24]). Since more detailed information is bound to contain sensitive data, users will most likely be unwilling to let developers collect such information. In addition, collecting large amounts of information would make it infeasible for users to sanitize the collected information by hand. To address this problem, some of these techniques suggest using an input minimization approach (*e.g.,* [6, 7, 35]) to reduce the number of failure-inducing inputs and, hopefully, eliminate some sensitive information. Input-minimization techniques, however, were not designed to specifically reduce sensitive inputs, so they can only eliminate sensitive data by chance. In order for techniques that leverage captured field information to become widely adopted and achieve their full potential, new approaches for addressing privacy and security concerns must be developed.

In this paper, we present a novel technique that addresses privacy and security concerns by sanitizing information captured from deployed applications. Our technique is designed to be used in conjunction with an execution capture/replay technique (*e.g.,* [4, 14]). Given an execution recording that contains a captured failure-inducing input $I = \langle i_1, i_2, \ldots i_n \rangle$ and terminates with a failure f, our technique replays the execution recording and leverages a specialized version of symbolic-execution to automatically produce I', a sanitized version of I, such that I' (1) still causes f and (2) reveals as little information about I as possible. A modified execution recording where I' replaces I can then be constructed and sent to the developers, who can use it to debug f.

It is, in general, impossible to construct I' such that it does not reveal any information about I while still causing the same failure f. Typically, the execution of f would depend on the fact that some elements of I have specific values (*e.g.,* i_1 must be 0 for the failing path to be taken). However, this fact does not prevent the technique from being useful in practice. In our evaluation, we found that the information revealed by the sanitized inputs was not sensitive and tended to be structural in nature (*e.g.,* a specific portion of the input must be surrounded by double quotes). Conversely, the parts of the inputs that were more likely to be sensitive (*e.g.,* values contained inside the double quotes) were not revealed (see Section 4).

To evaluate the effectiveness of our technique, we implemented it in a prototype tool, called CAMOUFLAGE, and carried out an empirical evaluation of 170 failure-inducing inputs for several real applications. In the evaluation, we investigated three research questions that are concerned with the feasibility, strength, and effectiveness of the approach. The results of the evaluation show that: (1) our approach is feasible in that, for each input that we considered, CAMOUFLAGE generated, in a matter of minutes, a sanitized version that reproduced the original failure; (2) the percentage of information revealed by the sanitized inputs ranged from ≈ 60% (in the worst case) to ≈ 2% (in the best case); and (3) even in the worst case, the sanitized inputs were unlikely to reveal sensitive information and could have been safely sent to developers. Although still preliminary, these results are promising and show that our approach can be both efficient and effective at sanitizing inputs that cause failures in real applications.

The contributions of this paper are:

- A novel technique for automatically generating sanitized versions of failure-inducing inputs.
- A prototype tool that implements our technique for Java applications.
- An extensive empirical study that demonstrates the feasibility and effectiveness of our technique.

The remainder of this paper is organized as follows: Section 2 provides an example that we used to motivate our technique. Section 3 describes our technique in detail. Section 4 presents our empirical evaluation. Section 5 presents related work and Section 6 presents our conclusions and possible future work.

2. MOTIVATING EXAMPLE

In this section, we provide an example that will be used in the remainder of the paper to illustrate our technique. Figure 1 shows the code for the example, which is an excerpt from a credit card processing utility that accepts Visa, American Express, and Discover credit cards. The program reads from the command line the credit card number to be processed and passes it to `isValidCardNumber`, which checks whether the provided number is valid using the Luhn formula (a simple checksumming algorithm). If the credit card number is valid, the program invokes `processCard`, which determines the type of the credit card number (*i.e.,* Visa, American Express, or Discover) by checking the number's prefix and processes the card accordingly.

Function `processCard` contains a fault that can cause the credit card processing utility to incorrectly handle certain credit card numbers. On October 1, 2006, Discover's prefix was changed from "650" to "65". Because line 20 of `processCard` was not updated to reflect this change, valid Discover card numbers that start with "65[1–9]", such as 6521 1065 6000 0061,[1] are not correctly processed and would cause an `UnknownCardType` exception to be thrown.

Although this program and fault are relatively simple to understand, failures caused by this fault are good examples of the type of scenario that our technique targets, for two reasons. First, such failures directly involve sensitive information (credit card numbers, in this case), which means that users would likely be unwilling to provide developers with the specific input that triggered the fault. Second, it would be difficult for commonly used approaches to provide a sanitized version of the input that still triggers the fault. In particular, input-minimization techniques would be likely to fail. Minimization techniques that attempt to find a subset of the inputs that causes the same failure, such as **ddmin** [35] or **delta** [6], will be unsuccessful because a valid credit card number must have 16 digits, so no minimization would be possible. Minimization techniques that perform alphabet normalization by substituting some portions of the input with a "don't care" value (*e.g.,* **tmin** [7]) would also likely fail, as most inputs generated in this manner will not satisfy the Luhn formula. Even constructing sanitized inputs by hand would be quite difficult, due to the difficulty of generating inputs that pass the Luhn check.

```
    boolean isValidCardNumber(String ccn) {
 1.   if(ccn.length() != 16) return false;
 2.   int sum = 0;
 3.   boolean alternate = false;
 4.   int i = ccn.length() - 1;
 5.   for (; i >= 0; i--) {
 6.     int n = mapChar(ccn.charAt(i));
 7.     if (alternate) {
 8.       n *= 2;
 9.       if (n > 9) n = (n % 10) + 1;
10.     }
11.     sum += n;
12.     alternate = !alternate;
13.   }
14.   return (sum % 10) == 0;
    }

    void processCard(String ccn) {
15.   if(ccn.startsWith("4"))
16.     //process Visa
17.   else if(ccn.startsWith("34")
18.           || ccn.startsWith("37"))
19.     //process American Express
20.   else if(ccn.startsWith("650"))
21.     //process Discover
22.   else
23.     throw new UnknownCardType(ccn);
    }

    int mapChar(char c) {
24.   return (c >= '0' && c <= '9') ? c-'0' : c-'A'+10;
    }

    void main(String[] args) {
25.   if(isValidCardNumber(args[0]))
26.     processCard(args[0]);
    }
```

Figure 1: Code excerpt for our motivating example.

3. AUTOMATIC SANITIZATION

Before discussing the details of our approach, we use Figure 2 to illustrate, intuitively, the goal of the approach and the context in which it operates. Given a program P with input domain ID, a failure f, and an input $I \in ID$ that causes f, there is in general a subset of the input domain, $\mathbb{I}_f \subseteq ID$, such that every input in $\mathbb{I}_f$ causes f.[2] In general, identifying $\mathbb{I}_f$ is impossible due to computability issues. However, under some assumptions that we discuss in Section 3.3, it is possible to identify a subset of $\mathbb{I}_f$, such that every input in this subset follows the same path as I and causes f.

To compute this subset, our approach uses a specialized version of symbolic execution [25]. Symbolic execution tech-

[1] Obviously, all credit card numbers used in this paper are presented for informational purposes only and should not be used in any other way.

[2] Note that this includes the extreme (and rare) case in which $\mathbb{I}_f$ is a singleton whose only element is I.

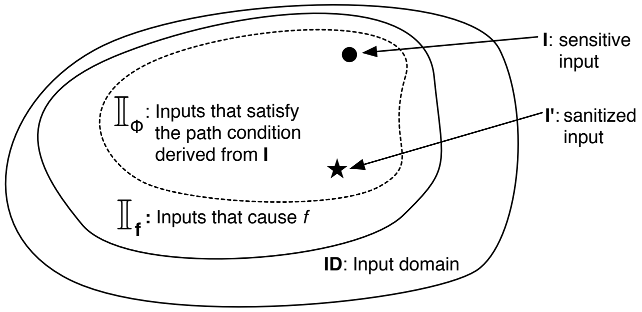

I: sensitive input
$\mathbb{I}_\Phi$: Inputs that satisfy the path condition derived from I
I': sanitized input
$\mathbb{I}_f$: Inputs that cause f
ID: Input domain

Figure 2: Intuitive view of a program domain.

niques execute a program using symbolic inputs so that, at each point in the computation, the state is expressed as a function of the input, and the conditions on the input for reaching the current location l are expressed as a conjunction of constraints called a *path condition*.

In our approach, we leverage this characteristic of symbolic execution to identify, given a specific input I, a subdomain of ID whose elements are inputs that cause the program to follow the same path as I. More precisely, our technique performs symbolic execution along the specific path of execution p caused by I; when failure f occurs at location l, the computed path condition, ϕ, identifies exactly the subdomain that we are looking for—the set of all inputs, including I, that cause p to be executed and f to occur at l. We call this set $\mathbb{I}_\phi$. (Note that, in general, the fact that an input satisfies ϕ does not necessarily imply that such input will follow the same path as I or, if it does, cause f. However, this tends to be the case in most situations, as we discuss in detail in Section 3.3.)

After computing ϕ, our approach generates a sanitized input I' by identifying a satisfying assignment for ϕ that is different from I. To do this, it leverages a solver for Satisfiability Modulo Theories (*i.e.,* a constraint solver), which, intuitively, chooses I' by selecting an input from $\mathbb{I}_\phi - I$.

The strength of our approach (*i.e.,* how well it prevents information about I from being revealed) depends on two related aspects. *First*, the solution identified by the solver, I', must be independent from I. We expect this assumption to be always satisfied, as the solver knows nothing about I when it solves ϕ. Moreover, most constraint solvers utilize some randomness in their search heuristics, so the selection of I' can be safely considered pseudo-random. *Second*, the subdomain identified by ϕ, $\mathbb{I}_\phi$, must be large enough to guarantee that an enumeration of the domain is impractical in a reasonable amount of time. (Because ϕ can be derived from I', just as it is derived from I, a trivially small domain allows for easily recovering I, which defeats the purpose of the technique.)

To strengthen our approach, particularly with respect to the second aspect, our approach specializes symbolic execution for our context of use by introducing *path condition relaxation.* Path condition relaxation is a suite of optimizations to basic symbolic execution that specialize constraint generation so as to increase the size of $\mathbb{I}_\phi$. Intuitively, path condition relaxation loosens overly restrictive constraints, thus allowing for a larger number of solutions, which increases the strength of the approach.

In the rest of this section, we discuss in detail how our technique performs symbolic execution and path condition relaxation.

Path condition:

```
; constraints from mapChar
ccn[0] ≥ '0' ∧ ccn[0] ≤ '9' ∧
...
ccn[15] ≥ '0' ∧ ccn[15] ≤ '9' ∧

; constraints from isValidCardNumber
((ccn[0] - '0') * 2) > 9 ∧ ((ccn[2] - '0') * 2) ≤ 9 ∧
((ccn[4] - '0') * 2) ≤ 9 ∧ ((ccn[6] - '0') * 2) > 9 ∧
((ccn[8] - '0') * 2) > 9 ∧ ((ccn[10] - '0') * 2) ≤ 9 ∧
((ccn[12] - '0') * 2) ≤ 9 ∧ ((ccn[14] - '0') * 2) > 9 ∧

((((((ccn[0] - '0') * 2) % 10) + 1) + ((ccn[1] - '0') + (((ccn[2]
- '0') * 2) + ((ccn[3] - '0') + (((ccn[4] - '0') * 2) + ((ccn[5]
- '0') + (((((ccn[6] - '0') * 2) % 10) + 1) + ((ccn[7] - '0') +
(((((ccn[8] - '0') * 2) % 10) + 1) + ((ccn[9] - '0') + (((ccn[10]
- '0') * 2) + ((ccn[11] - '0') + (((ccn[12] - '0') * 2) +
((ccn[13] - '0') + (((((ccn[14] - '0') * 2) % 10) + 1) + (ccn[15]
- '0')))))))))))))))) % 10) = 0 ∧

; constraints from processCard
ccn[0] ≠ '4' ∧
ccn[0] ≠ '3' ∧
ccn[0] = '6' ∧ ccn[1] = '5' ∧ ccn[2] ≠ '0'
```

Original input: `6510 2556 8418 3585`
Sanitized input: `6521 1065 6000 0061`

Figure 3: Path condition and sanitized input generated when the code in Figure 1 is run with `6521 1065 6000 0061` as input.

3.1 Symbolic Execution

As we mentioned in the previous section, our technique performs symbolic execution in parallel with a concrete execution, which is similar to what other recent techniques do (*e.g.,* DART [19], CUTE [30], EXE [11], and KLEE [10]). In our case, symbolic execution follows the path corresponding to a given failure-inducing input I rather than an arbitrary input chosen by the technique. In addition, our approach is only concerned with re-executing a specific, known to be feasible, path p (rather than exploring multiple, possibly infeasible paths), which helps the scalability of our approach.

To generate path conditions, our technique associates a symbolic variable v_k with every element of I and maintains a mapping from each symbolic variable to the location of the associated element (*e.g.,* the first character of the first command line argument or the i^{th} character of a particular file). For our example in Figure 1, for instance, each character of the command line argument would be associated with a unique symbolic variable: i_0 is associated with v_0, which is mapped to the first character of the first argument; i_1 is associated with v_1, which is mapped to the second character of the first argument; and so on. The mapping between variables and locations is used to generate I' from the solution provided by the constraint solver.

As the program executes, program statements are interpreted, and their effect determines (1) the symbolic state of the program, expressed in terms of the symbolic variables, and (2) the current path condition. As an example of a symbolic state, consider two subsequent instructions `S1`: $x = i + (j - 5)$ and `S2`: $y = x * 2$ where i is associated with a symbolic expression e and j is associated with symbolic variable v_1; the value of y in the symbolic state after $S2$'s execution would be the symbolic expression "$(e{+}(v_1{-}5)){*}2$".

Like traditional symbolic execution, our technique constructs path conditions incrementally, by appending a new constraint to the current path condition every time a predicate that depends on the symbolic state is executed (*i.e.,* every time a predicate uses one or more values that have an associated symbolic expression). The constraint encodes the condition on the symbolic variables under which the predicate evaluates in the same way as the concrete execution. To illustrate, we again use our motivating example. When line 24 is executed for the first time, c is associated with symbolic variable v_{15}, has the concrete value $'1'$, and is compared with character $'0'$. In this case, the predicate evaluates to `true` because $'1' \geq '0'$; therefore, the constraint "$v_{15} \geq '0'$" is appended to the path condition. Had the predicate evaluated to `false`, the constraint "$v_{15} < '0'$" would have been appended instead. Figure 3 shows the complete path condition generated by running the code for our motivating example with `6521 1065 6000 0061` as input.

After generating the path condition, our technique converts it into a representation that a constraint solver can handle. Currently, no constraint solver can handle the complete set of constructs that can be in a symbolic expression, such as multiplication of two symbolic expressions or bit-shifting by a symbolic amount. To handle these cases, our technique leverages a technique called *concretization* [19,30], which replaces the value of one or more symbolic variables involved in the problematic expressions with their corresponding concrete values. By making some symbolic values concrete, concretization reduces the domain identified by the path condition being solved, thus reducing the amount of information that can be sanitized. Nevertheless, concretization is typically necessary to make the approach practical. On the positive side, constraint solvers are constantly improving, so the number of situations that require concretization is likely to decrease over time.

3.2 Path Condition Relaxation

As we mentioned at the beginning of Section 3, path condition relaxation consists of several optimizations that modify the way path conditions are generated to increase the number of solutions for the computed conditions. There are many situations in which path condition construction can be optimized. In the following, we describe the four optimizations that our technique currently uses.

Array inequality. Typically, a comparison between two arrays is performed by iterating over the arrays and performing a pairwise comparison between corresponding elements. In traditional symbolic execution, the result of each comparison would be recorded as a constraint in the path condition. Therefore, only inputs that cause every comparison to evaluate the same way as the observed execution would satisfy the path condition. For example, assume that $a = [1, 2, 3]$, $b = [1, 2, 4]$, a's elements are associated with symbolic expressions e_1, e_2, and e_3, and b's elements are associated with symbolic expressions e_4, e_5, and e_6. Checking the equality of these arrays would add the constraints "$e_1 = e_4$", "$e_2 = e_5$" and "$e_3 \neq e_6$" to the path condition.

The key intuition behind our optimization of array comparisons is that such comparisons are essentially atomic operations. Therefore, when arrays are not equal, our technique can replace the constraints that encode the individual comparisons with a constraint that simply requires that at least one comparison evaluates to `false` (*i.e.,* at least one element is different). For the previous example, the constraint "$v_1 \neq v_3 \vee v_2 \neq v_4 \vee v_3 \neq v_6$" would be added. All variable assignments that satisfy the original constraints also satisfy the relaxed one, but this latter is also satisfied by many other assignments (*e.g.,* $a = [2, 2, 3]$, $b = [1, 2, 4]$).

Multi-clause conditionals. In many languages, commonly used Boolean operators such as "and" and "or" are evaluated with short-circuit or minimal evaluation semantics; only the minimal amount of evaluation is done to determine the value of the expression. For example, consider the conditional "`if`$(i_1 > 5 \parallel i_2 > 5)$". If i_1's value is 6, then "$i_2 > 5$" will not be evaluated, as the outcome of the condition is known after the evaluation of "$i_1 > 5$". Because the rest of the conditional is not evaluated, the path condition will only include the constraint "$v_1 > 5$". Like for array inequalities, such path conditions exclude a large number of assignments that would cause the conditional to evaluate to the same value (*e.g.,* $a = 0$, $b = 6$).

To generate relaxed path conditions for multi-clause conditionals, our technique generates constraints that encode all clauses in the conditional, not just those evaluated at runtime. For example, if the conditional "`if`$(i_1 > 5 \parallel i_2 > 5)$" were to evaluate to `true`, our technique would generate the constraint "$v_1 > 5 \vee v_2 > 5$". Conversely, if the conditional were to evaluate to `false`, the constraint "$v_1 < 5 \wedge v_2 < 5$" would be generated. Clauses joined by "and" and conditionals comprised of more than two clauses are handled in a similar manner.

Switch statements. Switch statements are similar to multi-clause conditionals in that multiple values can cause the switch to jump to the same target (*i.e.,* for default targets or case statements that immediately fall through to their successor). If the default branch is taken, our technique generates a constraint of the form "$v_i \neq c_1 \wedge \ldots \wedge v_i \neq c_n$", where v_j is the symbolic expression associated with the value compared by the switch statement, and each c is the value of one of the case statements inside the switch; the constraint forces the value of v_i to be different from all of the values in the case statements, but does not further restrict the range of values that can be chosen. Conversely, if one of the cases of the switch statement is taken, the technique generates a constraint of the form "$v_i = c_1 \vee \ldots \vee v_i = c_n$", where v_i is again, the symbolic expression associated with the value compared by the switch statement, and each c is one of the values of the cases that branch to the same target.

Array reads. Array accesses are a another case in which concretization is typically performed. For example, assume that $a = [2, 0, 1]$, and that x has the value "0" and is associated with symbolic expression e when statement "‘`if`$(a[x] > 0)$" is executed. For this statement, e would be concretized to "0" to ensure that the same path is followed.

Similar to the cases we discussed previously, concretizing in this case is unnecessarily restrictive, as multiple values in an array can satisfy the same condition. In our previous example, for instance, the values "2" at index 0 and "1" at index 2 will both cause the conditional to evaluate to `true`, which means that e can be equal to either "2" or "0". To generate path conditions that take this possibility into account, our technique uses an approach similar in nature to the one proposed by Elkarablieh and colleagues [18]. Essentially, our technique encodes a snapshot of the contents of the array into the path condition as sequence of ternary-expressions. Continuing with the previous example, the technique would generate the constraints "$((e == 0)\ ?\ 2 : (e == 1)\ ?\ 0 : 1) > 0 \ \wedge\ e \geq 0 \ \wedge\ e < 3$". These constraints ensure that the value of e is within the bounds of a, but otherwise allow it to be any value that satisfies the conditional.

3.3 Assumptions

Our technique is based on the assumption that any input that satisfies path condition ϕ not only follows the same path as the original failure-inducing input I, but also results in the same failure f. In practice, this requires that the necessary conditions for f are encoded in ϕ. Intuitively, the only cases in which this assumption does not hold are non-determinism and *implicit checks*.

If the program being considered is non-deterministic, our technique may not be able to generate sanitized inputs that reproduce the failure because it cannot guarantee that events such as thread switches always occur in the same order. Note that this problem is common to all debugging-related techniques and not specific to our approach. Like for these techniques, the issue could be addressed by leveraging a capture/replay technique that supports deterministic replay (*e.g.,* [4]). (As we stated in the Introduction, our technique is meant to be used in conjunction with a technique for execution capture and replay.)

Implicit checks are checks that are performed by an entity that is external to the application and, thus, are not observable by the symbolic execution. Two typical examples of implicit checks are checks performed by the underlying runtime system (*e.g.,* checks that may result in a division-by-zero or out-of-memory error) and checks performed by a human oracle (*e.g.,* a tester that classifies an execution as failing because the outcome produced is different from the expected one).

Because these checks are not performed by the application, they would not be included as constraints in the path condition ϕ, and an input that satisfies ϕ may fail to reproduce f. Although this issue exists, we believe it is of limited relevance in most cases, for several reasons. First, although (some types of) implicit checks occur frequently, we expect that the majority of them will be irrelevant because, as confirmed by our evaluation, they constraint variables that are not directly or indirectly related to the failure. Therefore, in the worst case, our technique could simply ignore cases for which the sanitized input cannot reproduce f and focus on the remaining failures. Second, we can automatically account for implicit checks that occur within the runtime system by making them explicit. To account for division-by-zero errors, for instance, our technique could add an explicit check of the denominator's value every time a division is encountered. Third, in the case of checks that cannot be automatically handled, such as human or external checks, we could either ignore the corresponding failures, as discussed above, or rely on some form of built-in oracle. (This is analogous to relying on an accurate automated oracle, as many automated debugging techniques do.)

4. EVALUATION

To evaluate our technique we implemented it in a prototype tool, called CAMOUFLAGE, and investigated the following research questions:

RQ1: Feasibility—Can our approach generate, in a reasonable amount of time, sanitized inputs that reproduce the original failure?

RQ2: Strength—How much information about the original failure-inducing inputs is revealed by the approach?

RQ3: Effectiveness—Are the sanitized inputs generated by our approach safe to send to developers?

Note that RQ2 provides an objective assessment of our technique; it does not make any assumptions about whether the revealed information is actually sensitive. Conversely, RQ3 does take into account whether the information that is revealed is indeed sensitive.

The remainder of this section discusses CAMOUFLAGE, our subjects, and our experimental protocol and results.

4.1 Prototype Tool

Our CAMOUFLAGE tool is a prototype implementation of our technique for applications written in the Java language. It consists of two separate components: the constraint generator and the input sanitizer. (We consider the capture/replay tool that would provide inputs to be sanitized to CAMOUFLAGE as an external component.) The current implementation of the constraint generator is an extension to NASA's explicit state software model checker for Java software: Java PathFinder (JPF) (`http://javapathfinder.sourceforge.net/`). We chose JPF as the basis for the constraint generator because it has many capabilities that simplify our implementation (*e.g.,* bytecode overloading and uncaught exception handling) and has been successfully extended with features similar to the ones we need (*i.e.,* concolic execution [23] and symbolic execution [32]). In fact, we were able to reuse some portions of the concolic execution extension in our tool.

To assign symbolic variables to an application's inputs, we use JPF's method interception capabilities to wrap all native methods in the `java.io` package. Because, ultimately, all file and network inputs are read by these methods, CAMOUFLAGE can easily associate a symbolic variable with every input read from these sources. To handle other sources of input, we also wrap the `main` method (to handle command line arguments) and the appropriate methods for reading environment variables and system properties. By default, CAMOUFLAGE assumes that all inputs are sensitive. However, as a convenience, it also allows users to specify that inputs read from specific sources should not be associated with a symbolic variable. This feature is useful, for example, in cases where it is known that inputs read from certain files or network streams are not sensitive and do not need to be sanitized. To implement our specialized path condition generation (see Section 3.2), we use JPF's bytecode overloading facilities to replace each Java bytecode with a modified version that replicates the instruction's original semantics, but also performs the necessary steps for generating path conditions. Finally, to identify when failures occur, we use JPF's VMListener interface to intercept uncaught exceptions.

When the execution reaches the point of failure, and the failure occurs, the constraint generator writes the recorded path condition to disk. In addition, it also stores a set of constraints that prevent the constraint solver from selecting the original input. These additional constraints are necessary; we have encountered instances, albeit rarely, where without these constraints, the solver happened to select inputs that were unnecessarily similar to the original inputs (*i.e.,* portions of the sanitized input were identical to the corresponding portions of the original input, even though other values could have been chosen). Note that we add these additional constraints as "discardable" constraints that can be ignored if the constraint solver cannot satisfy them. If these constraints could not be ignored, there may be cases where their presence would make the path condition unsatisfiable (*e.g.,* when portions of an input *must* have a given value for the failure of interest to occur). Using discardable constraints allows CAMOUFLAGE to handle these situations.

The input sanitizer is implemented as a set of Ruby scripts and works as follows. First, it transforms the constraints produced by the constraint generator into a format understood by the constraint solver. Then, it invokes the constraint solver to find a solution for the constraints. Finally, it transforms the solution provided by the constraint solver into a concrete input that can be sent to developers. As our constraint solver, we choose YICES [16] because of its support for bit vector operations and discardable constraints. Among the constraint solvers that we are aware of, it is the only one to support both of these features. Using bit vectors for symbolic variables allows our implementation to handle bit shifts and masks, which are commonly used in the Java libraries. However, using bit vectors does have one drawback: currently, no constraint solver, including YICES, supports floating point arithmetic on bit vectors. This means that CAMOUFLAGE does not support associating a symbolic variable with floats or doubles.

4.2 Subjects

The goal of our technique is to generate sanitized inputs that cause the same failures as the original input while revealing as little information as possible. To suitably evaluate our technique with respect to this goal, we selected applications with known faults that process information that can be considered private or sensitive: `NanoXML` (16 faults), which is available from the Software-artifact Infrastructure Repository (SIR) [15], a Java version of `printtokens` (2 faults), whose original C implementation is also available from SIR; the address book component of the `Columba` email client version 1.4 (1 fault) (`http://www.columbamail.org`); and version 1.0 of `htmlparser` (1 fault) (`http://htmlparser.sourceforge.net`). For each fault, we selected multiple failure-inducing inputs. For `NanoXML` and `printtokens`, we used the failure-inducing inputs provided with the two applications. For `Columba` and `htmlparser`, we constructed representative inputs by hand. In total, we used 170 failure-inducing inputs that range in size from several hundred bytes to over 5 megabytes.

4.3 RQ1: Feasibility

The goal of our first research question is to assess whether the amount of time needed to generate sanitized inputs is reasonable and whether the sanitized inputs reproduce the original failure. To generate the data necessary for investigating these questions, we proceeded as follows: for each failure-inducing input, we used CAMOUFLAGE to run the application and generate a sanitized version of such input. In addition, we recorded two measurements: (1) the amount of time needed by CAMOUFLAGE to generate the path condition and (2) the amount of time needed by the constraint solver to solve the generated path condition.

The top-half of Figure 4 presents a bar chart that shows, for each fault, the average amount of time CAMOUFLAGE needed to generate path conditions. The bottom-half of the figure shows the average amount of time needed by the constraint solver to solve the generated path conditions. As the figure shows, the amount of time needed to generate path conditions ranges from an average of 162 seconds (for `printtokens`) to an average of 533 seconds (for `htmlparser`). The amount of time needed to solve the path conditions ranges from an average of 0.1 seconds (for `printtokens`) to an average of 15.7 seconds (for `Columba`). For all of the failure-inducing inputs that we considered, CAMOUFLAGE was able to generate a sanitized version in less than 10 minutes. Because CAMOUFLAGE is designed to run off-line, during idle periods when free cycles are available (*e.g.,* overnight), the approach is clearly practical. Users will only experience the overhead caused by the capture/replay technique used, which have been shown to be in the single digits for modern approaches [4, 14].

To determine whether the sanitized inputs reproduce the original failures, we executed our subject applications with their sanitized inputs and manually inspected the outcomes. We found that all 170 sanitized inputs produced by CAMOUFLAGE successfully reproduced the original failure.

4.4 RQ2: Strength

To assess the strength of the sanitization performed by CAMOUFLAGE, we used the following two metrics: bits of information revealed and residue. The first metric, *bits of information revealed*, is a standard entropy measure that has been used in related work [12, 33]. Intuitively, it measures how much information is revealed by the technique by calculating how many inputs satisfy the path condition (*i.e.,* the number of inputs in $\mathbb{I}_\phi$). In general, a sanitized input reveals $\sum_{i \in I'} |log_2(x_i)|$ bits of information about I, where x_i is the number of solutions to the constraints involving i divided by the size of i's input domain. For example, assume that i'_0 is an 8-bit character (*i.e.,* its input domain contains 256 values) and that 5 of the 256 possible values satisfy the constraints on i'_0. In this case, i'_0 reveals approximately 5.76 of the 8 total bits of information about i_0. Because computing x_i exactly is difficult and expensive when constraints involve multiple input elements, we chose to use an algorithm by Martin that quickly provides an accurate over-approximation of x_i [28].

The bits-of-information-revealed metric provides a good starting point for assessing the strength of the sanitization. However, its results can be misleading. For example, it is possible to decrease the amount of bits revealed while large portions of the input remain unchanged. To illustrate this situation, consider a program that reads 10 characters as input. Assume that the constraints on each of the last 5 characters have 10 possible solutions, while the first 5 characters must remain the same. If the number of possible solutions for the second 5 characters is increased from 10 to 200, the amount of information revealed decreases from 63.3 bits to 41.7 bits. This decrease correctly indicates that it is now more difficult to recover the original input, but it fails to indicate that half of the input is unchanged, a fact that may be important, especially if the first half of the input is more sensitive than the second half.

Our second metric, *residue*, addresses this shortcoming. Residue is essentially the number of inputs that remain unchanged after sanitization. For the example mentioned in the previous paragraph, the percentage of residue would not change if the number of possible solutions for the second 5 characters increased from 10 to 200, thus indicating that sanitization may not have been as effective as the bits of information revealed metric would suggest. By using both metrics, we can assess the strength of the sanitization performed by CAMOUFLAGE from multiple perspectives and better judge how much information about the failure-inducing inputs is revealed by their sanitized versions.

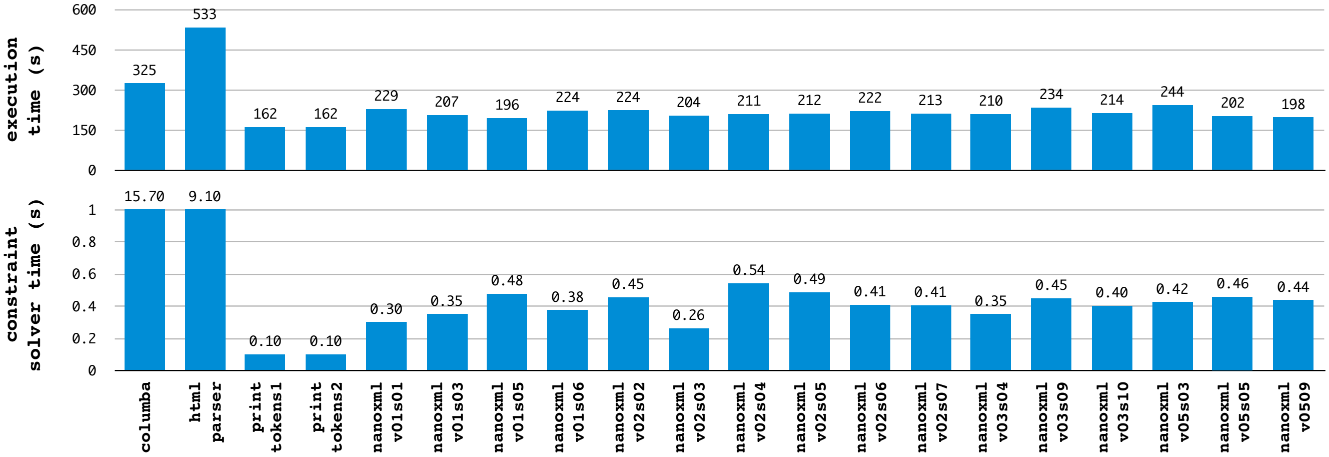


Figure 4: Bar charts showing, for each fault, the average amount of time needed to execute the subject and generate the corresponding path condition (top) and average amount of time needed for the constraint solver to find a solution (bottom).

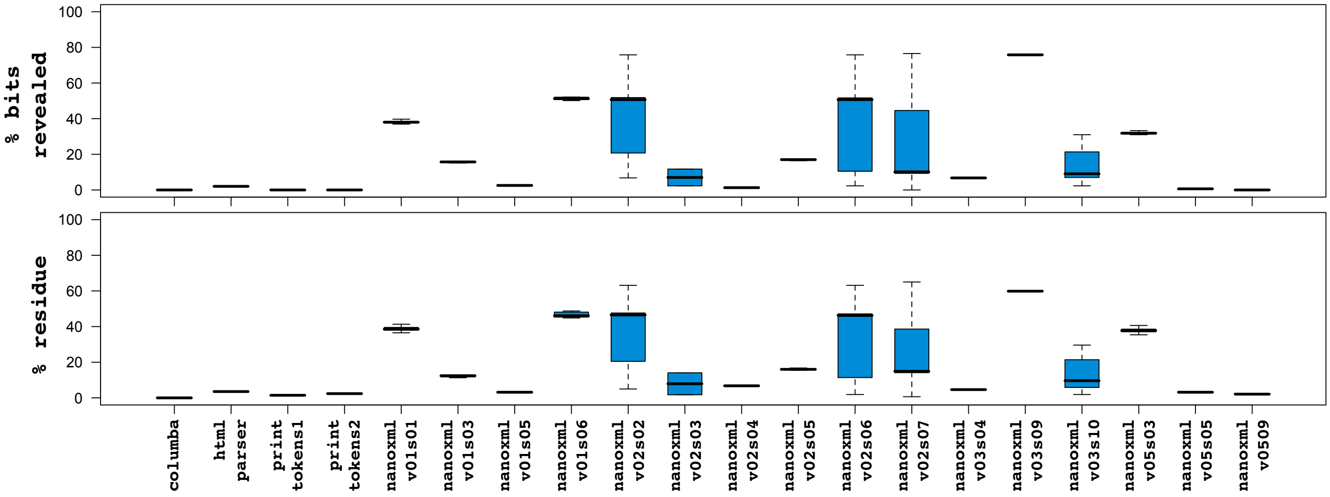
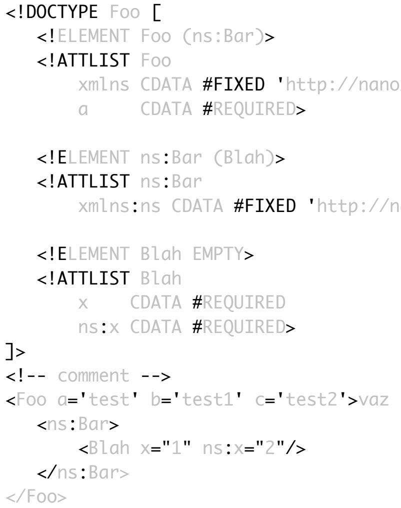


Figure 5: Box plots showing, for each fault, the bits of information revealed as a percentage of the total number of bits in the input (top) and the percentage of residue (bottom) that remains after sanitization.

Figure 5 presents two box-and-whisker plots that show, for each fault and failure-inducing input, the bits of information revealed by the sanitized input as a percentage of the total number of bits in the failure-inducing input (top) and the percentage of residue in the sanitized input (bottom). For the subjects we considered, the average percentage of bits of information revealed ranges from 2.3% to 76.5%, with an overall average of 30.6% and the average percentage of residue ranges from 1.5% to 65%, with an overall average of 30%. Although these results confirm that, in general, it is impossible to generate sanitized inputs that reveal no information about the original inputs, they are also encouraging; the majority of sanitized inputs produced by CAMOUFLAGE only reveal a limited amount of information. (Moreover, as the results discussed in the next section show, the information revealed is unlikely to be sensitive.) The results also suggest that the strength of the sanitization performed by CAMOUFLAGE depends not only on the subject application, but also on the specific input and can vary widely even among different inputs that trigger the same fault.

4.5 RQ3: Effectiveness

The results of RQ2's investigation provide an objective measure of the sanitation performed by CAMOUFLAGE. However, without considering whether the revealed information is actually sensitive, it is difficult to accurately assess if sanitized inputs can safely be sent to developers. Performing such an assessment is the goal of the study addressing RQ3. In this study, we conducted an in-depth, qualitative assessment of all the sanitized inputs generated by CAMOUFLAGE that takes into account whether the revealed information is sensitive. To make this determination, we manually inspected each failure-inducing input and its sanitized version. (As the discussion of the specific sanitization cases in the rest of this section will show, for the subjects we considered the distinction between sensitive and not sensitive was fairly clear-cut.) For all 170 sanitized inputs, we found that they did not reveal any information that we believe to be sensitive. In the rest of this section, we provide a detailed description of our analysis for three sanitized inputs: one for



```
<!DOCTYPE Foo [
   <!ELEMENT Foo (ns:Bar)>
   <!ATTLIST Foo
       xmlns CDATA #FIXED 'http://nanoxml.n3.net/bar'
       a     CDATA #REQUIRED>

   <!ELEMENT ns:Bar (Blah)>
   <!ATTLIST ns:Bar
       xmlns:ns CDATA #FIXED 'http://nanoxml.n3.net/bar'>

   <!ELEMENT Blah EMPTY>
   <!ATTLIST Blah
       x    CDATA #REQUIRED
       ns:x CDATA #REQUIRED>
]>
<!-- comment -->
<Foo a='test' b='test1' c='test2'>vaz
   <ns:Bar>
       <Blah x="1" ns:x="2"/>
   </ns:Bar>
</Foo>
```

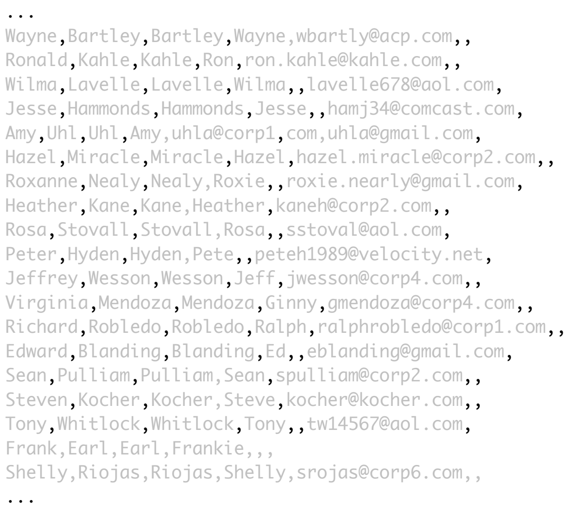
Figure 6: Failure-inducing input for NanoXML.

NanoXML, one for the address book component of Columba, and one for htmlparser. We chose to present these inputs because, among the failure-inducing inputs for each application, they have the highest percentage of bits of information revealed and residue. Consequently, they are the most likely to actually reveal sensitive information.

Figures 6, 7, and 8 show representations of the portions of the original inputs that can and cannot be changed (*i.e.,* residue) for the inputs we are presenting. In these figures, portions of the inputs that can be changed are colored gray, while portions that cannot be changed are colored black.

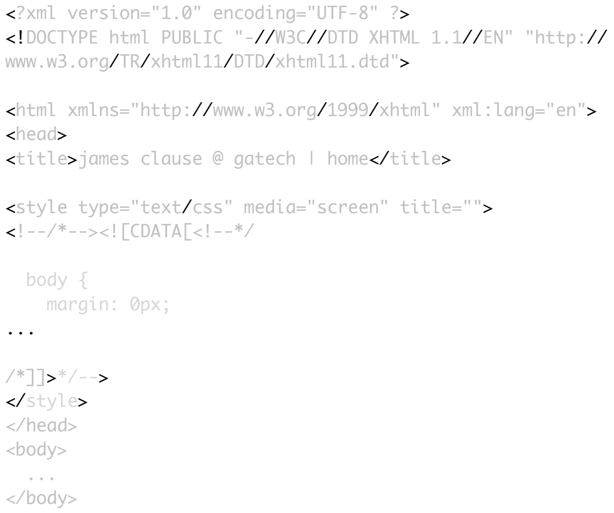
NanoXML. The input for NanoXML shown in Figure 6 is an XML file available from SIR repository. The fault triggered by this input causes NanoXML to incorrectly handle closing tags. As the figure shows, the portions of this file that cannot be changed do not contain any sensitive information. The literals "DOCTYPE", "ATTLIST", and "FIXED" are keywords of the language used to specify document type definitions, and NanoXML specifically checks for their presence. Similarly, the angle brackets, exclamation points, hyphens, double quotation marks, backslashes, and equals signs that cannot be changed are necessary because they define the structure of the XML document. Conversely, portions of the input that are likely to contain sensitive information, such as XML tag names, attribute values, and tag bodies, can all be changed without preventing the modified input from reproducing the failure. Therefore, although a relatively large percentage of the file cannot be changed (≈ 65%), we can consider the input to be sanitized because it contains, to the best of our knowledge, no real sensitive information.

Columba. The input for Columba shown in Figure 7 is a comma-separated-value file of contact information. The entries in each row are a contact's first name, last name, sort key, nickname, work phone, and home phone. The fault that this file triggers is in a section of Columba that handles the email portions of each row. Columba assumes that each contact has either a work or a home email address. If this assumption is violated, as it is by the second-to-last row in the part of the input shown in Figure 7, an exception is thrown. The results of sanitizing this input are similar to the results of sanitizing the input for NanoXML; the structural elements of the file (*i.e.,* the commas that separate the individual fields) cannot be changed, but the non-structural elements (*i.e.,* each contact's first name, last name, and so on) can all be changed. Consequently, we can conclude that the input produced by CAMOUFLAGE is sanitized (*i.e.,* contains no sensitive information) and can be safely sent to developers.



```
...
Wayne,Bartley,Bartley,Wayne,wbartly@acp.com,,
Ronald,Kahle,Kahle,Ron,ron.kahle@kahle.com,,
Wilma,Lavelle,Lavelle,Wilma,,lavelle678@aol.com,
Jesse,Hammonds,Hammonds,Jesse,,hamj34@comcast.com,
Amy,Uhl,Uhl,Amy,uhla@corp1,com,uhla@gmail.com,
Hazel,Miracle,Miracle,Hazel,hazel.miracle@corp2.com,,
Roxanne,Nealy,Nealy,Roxie,,roxie.nearly@gmail.com,
Heather,Kane,Kane,Heather,kaneh@corp2.com,,
Rosa,Stovall,Stovall,Rosa,,sstoval@aol.com,
Peter,Hyden,Hyden,Pete,,peteh1989@velocity.net,
Jeffrey,Wesson,Wesson,Jeff,jwesson@corp4.com,,
Virginia,Mendoza,Mendoza,Ginny,gmendoza@corp4.com,,
Richard,Robledo,Robledo,Ralph,ralphrobledo@corp1.com,,
Edward,Blanding,Blanding,Ed,,eblanding@gmail.com,
Sean,Pulliam,Pulliam,Sean,spulliam@corp2.com,,
Steven,Kocher,Kocher,Steve,kocher@kocher.com,,
Tony,Whitlock,Whitlock,Tony,,tw14567@aol.com,
Frank,Earl,Earl,Frankie,,,
Shelly,Riojas,Riojas,Shelly,srojas@corp6.com,,
...
```

Figure 7: Failure-inducing input for Columba's address book component.



```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://
www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>james clause @ gatech | home</title>

<style type="text/css" media="screen" title="">
<!--/*--><![CDATA[<!--*/

  body {
    margin: 0px;
...

/*]]>*/-->
</style>
</head>
<body>
  ...
</body>
```

Figure 8: Failure-inducing input for htmlparser.

HtmlParser. The input for htmlparser shown in Figure 8 is an HTML file taken from the website of one of the authors. The fault that this input triggers is in the tag-processing portion of htmlparser, which scans for pairs of angle brackets and backslashes. This version of htmlparser incorrectly handles several angle brackets around the CDATA portion of the file, which causes a mismatch between opening and closing brackets and leads, ultimately, to an exception being thrown. For this input, the only parts that cannot be changed are the angle brackets and backslashes, which are explicitly matched by the tag parser. Again, the portions of the input that are most likely to be sensitive—the contents of the web page and the style sheet—have all been changed.

Overall conclusions. For the three failure-inducing inputs that we presented above, and the additional 167 inputs that we considered in our evaluation, CAMOUFLAGE was always able to sanitize the inputs by removing all of the portions of the inputs that we considered to be sensitive. These results are encouraging because they provide initial, but strong, evidence that CAMOUFLAGE can generate sanitized failure-inducing inputs that could be safely sent to developers.

4.6 Threats to Validity

Because we used a limited number of subjects and faults, our results may not generalize. However, both the subjects and the faults we considered are real and representative of the type of situations we expect to encounter in practice. Therefore, we believe that these results, albeit preliminary, are promising and warrant additional experimentation with more varied and larger subjects.

5. RELATED WORK

Currently, only a few techniques directly address the problem of eliminating sensitive information from captured data.

The technique most closely related to ours is the one proposed by Castro and colleagues [12], which is also based on symbolic execution. The main advantage of our technique over theirs is the use of a customized symbolic execution algorithm, rather than a traditional one, which enables our approach to generate larger sub-domains and should ultimately result in a more effective sanitization. Although we cannot perform a direct comparison of the two approaches because their implementation (1) works for x86 binaries and (2) is not publicly available, we performed a study to get an initial assessment of the difference in effectiveness between the two techniques. To do this, we developed a Java implementation of Castro and colleagues' technique using JPF's symbolic execution engine and compared its performance with the performance of CAMOUFLAGE (in terms of time needed to generate sanitized inputs, bits of information revealed, and residue) when run on our set of subjects and failure-inducing inputs. The results of this comparison show that CAMOUFLAGE required only slightly more time to generate sanitized inputs, and, on average, the sanitized inputs generated by CAMOUFLAGE revealed 30% less bits of information and contained 40% less residue. With the caveat of a potential implementation bias, these results provide clear evidence that our technique can be more effective at sanitizing inputs than Castro and colleagues' technique.

Broadwell and colleagues' SCRASH tool uses a form of secure information flow (dynamic tainting) to identify where sensitive information is stored inside a crash dump [9]. During an execution, an initial set of of data is marked as sensitive. As the execution progresses, any data that is derived from this initial set is also marked as sensitive. Finally, when a crash occurs, any data that is marked as sensitive is excluded from the crash dump that is sent to developers. The main practical limitation of this approach is the difficulty in identifying the initial set of sensitive data—it is unreasonable to expect users to perform this step, and relying on the application's developers is equivalent to trusting them with access to the sensitive data. Furthermore, unlike our technique, SCRASH does not attempt to sanitize sensitive data, but simply avoids sending it to the developers, which would result in a loss of potentially useful information on the developers' side. In addition, their technique is performed on-line and, unlike our technique, may subject users to high runtime overheads.

Wang and colleagues propose an approach, PANALYST [33], that aims to reconstruct failure-inducing inputs on developers' machines by using a combination of dynamic taint analysis, symbolic execution, and collection of answers to questions sent to a client running on the user's machine. Answers provided by the client determine which direction the symbolic execution takes when encountering branches that depend on sensitive information and what values are read or written by memory accesses through sensitive pointers. The client will answer all questions that do not involve sensitive information, but will only disclose up to a predetermined amount of sensitive information. Like for SCRASH, the main practical limitation of this technique is the difficulty in identifying which information is sensitive and how much sensitive information is safe to send to developers. In addition, there are also technical limitations that may prevent the approach from scaling beyond the stateless packet processing application on which it has been evaluated.

In addition to the techniques that are directly related to ours, there is also a large body of work that is concerned with anonymizing data sets (*e.g.,* databases or spreadsheets) before they are released to the public (*e.g.,* [31, 34, 36]). These approaches try to maintain statistical properties of the data (*e.g.,* the distribution of ages across a population) while preventing users of the data from uniquely identifying a specific record (*e.g.,* the age of a specific individual). Typically, this is accomplished by merging data (*e.g.,* ages 0–18 are grouped together) or by adding random noise to the data. Because the conditions for reproducing a failure are typically very specific, these approaches are not suitable for our scenario.

White-box dynamic test generation and fuzzing techniques (*e.g.,* [10, 11, 19, 30, 32]) are also tangentially related to our technique. Instead of solving path conditions to obtain a new set of inputs to reach a known failure, they iteratively generate, negate, and solve path constraints to explore multiple execution paths.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a novel technique for sanitizing inputs that cause failures. Given a failure-inducing input, our technique (1) identifies an input set that includes this input together with other inputs that induce the same failure and (2) selects an input different from the initial one from this set. To do this, our technique leverages a specialized version of symbolic execution and various optimizations that aim to increase the size of the failure-revealing input set (so as to increase the effectiveness of the sanitization).

We also presented CAMOUFLAGE, a prototype implementation of our approach for Java programs, and an empirical evaluation of CAMOUFLAGE on 170 failure-inducing inputs for several real applications. The results of the evaluation show that our approach is feasible and effective. For each failure-inducing input that we considered, CAMOUFLAGE was able to generate a sanitized version that reproduced that original failure in less than 10 minutes; an amount of time that is well within the length of typical idle periods on a user's machine. Moreover, manual investigation of the sanitized inputs shows that they do not reveal any potentially sensitive information contained in the original inputs, and could therefore be safely sent to developers.

As future work, we will investigate additional metrics for quantifying the strength of the sanitization. Bits of information revealed and residue constitute a useful starting point, but, as we mentioned in the paper, they fail to account for all aspects of privacy loss. Most importantly, they do not consider the relative sensitivity of different parts of the inputs. In addition, we believe that for many users these metrics would be difficult to use effectively, as they provide no indication of what is an acceptable percentage of bits of information revealed or residue. As our evaluation shows, even when a sanitized input reveals a relatively large amount of information, it may still be safe to send to developers.

7. REFERENCES

[1] Apport - Automatic crash reports, September 2009. https://wiki.ubuntu.com/Apport.

[2] Privacy Statement for the Microsoft Error Reporting Service, September 2009. http://oca.microsoft.com/en/dcp20.asp.

[3] Technical Note TN2123: CrashReporter, September 2009. http://developer.apple.com/technotes/tn2004/tn2123.html.

[4] The Amazing VM Record/Replay Feature in VMware Workstation 6, September 2009. http://blogs.vmware.com/sherrod/2007/04/the_amazing_vm_.html.

[5] Windows Error Reporting: Getting Started, September 2009. http://www.microsoft.com/whdc/maintain/StartWER.mspx.

[6] Delta, September 2009. http://delta.tigris.org/.

[7] tmin: Fuzzing Test Case Optimizer, September 2009. http://code.google.com/p/tmin/.

[8] Apple. Apple Customer Privacy Policy, September 2009. http://www.apple.com/legal/privacy/.

[9] P. Broadwell, M. Harren, and N. Sastry. Scrash: A system for Generating Secure Crash Information. In *Proceedings of the 12th Conference on USENIX Security Symposium*, pages 19–19, 2003.

[10] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224, 2008.

[11] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 322–335, 2006.

[12] M. Castro, M. Costa, and J.-P. Martin. Better Bug Reporting with Better Privacy. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 319–328, 2008.

[13] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani. HOLMES: Effective Statistical Debugging via Efficient Path Profiling. In *Proceedings of the 31st International Conference on Software Engineering*, pages 34–44, 2009.

[14] J. Clause and A. Orso. A Technique for Enabling and Supporting Debugging of Field Failures. In *Proceedings of the 29th IEEE and ACM SIGSOFT International Conference on Software Engineering*, pages 261–270, 2007.

[15] H. Do, S. G. Elbaum, and G. Rothermel. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[16] B. Dutertre and L. de Moura. The YICES SMT Solver. http://yices.csl.sri.com/tool-paper.pdf.

[17] S. Elbaum and M. Diep. Profiling Deployed Software: Assessing Strategies and Testing Opportunities. *IEEE Transactions on Software Engineering*, 31(4):312–327, 2005.

[18] B. Elkarablieh, P. Godefroid, and M. Y. Levin. Precise pointer reasoning for dynamic test generation. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, pages 129–140, 2009.

[19] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[20] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of the Network and Distributed System Security Symposium*, 2008.

[21] D. Hilbert and D. Redmiles. An Approach to Large-scale Collection of Application Usage Data Over the Internet. In *Proceedings of the 20th International Conference on Software Engineering*, pages 136–145, 1998.

[22] D. M. Hilbert and D. F. Redmiles. Extracting Usability Information from User Interface Events. *ACM Computing Surveys*, 32(4):384–421, Dec 2000.

[23] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kieżun. jFuzz: A Concolic Whitebox Fuzzer for Java. In *Proceedings of the First NASA Formal Methods Symposium*, pages 121–125, 2009.

[24] L. Jiang and Z. Su. Context-aware Statistical Debugging: From Bug Predictors to Faulty Control Flow Paths. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, pages 184–193, 2007.

[25] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[26] B. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, 2004.

[27] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable Statistical Bug Isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 15–26, 2005.

[28] J.-P. Martin. Upper and lower bounds on the number of solutions. Technical Report MSR-TR-2007-164, Microsoft Research, 2007.

[29] C. Pavlopoulou and M. Young. Residual Test Coverage Monitoring. In *Proceedings of the 21st International Conference on Software Engineering*, pages 277–284, 1999.

[30] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference / 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 263–272, 2005.

[31] V. S. Verykios, E. Bertino, I. N. Fovino, L. P. Provenza, Y. Saygin, and Y. Theodoridis. State-of-the-art in Privacy Preserving Data Mining. *ACM SIGMOD Record*, 33(1):50–57, 2004.

[32] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test Input Generation with Java PathFinder. *SIGSOFT Software Engineering Notes*, 29(4):97–107, 2004.

[33] R. Wang, X. Wang, and Z. Li. Panalyst: Privacy-aware Remote Error Analysis on Commodity Software. In *Proceedings of the 17th USENIX Security Symposium*, pages 291–306, 2008.

[34] Z. Yang, S. Zhong, and R. N. Wright. *Privacy-Preserving Queries on Encrypted Data*, volume 4189 of *Lecture Notes in Computer Science*, pages 479–495. 2006.

[35] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[36] S. Zhong, Z. Yang, and R. N. Wright. Privacy-Enhancing K-Anonymization of Customer Data. In *Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 139–147, 2005.