Replayer: Automatic Protocol Replay by Binary Analysis

James Newsome, David Brumley, Jason Franklin, Dawn Song*
Carnegie Mellon University
Pittsburgh, PA, USA
{jnewsome, dbrumley, jfranklin, dawnsong}@cmu.edu

ABSTRACT

We address the problem of replaying an application dialog between two hosts. The ability to accurately replay application dialogs is useful in many security-oriented applications, such as replaying an exploit for forensic analysis or demonstrating an exploit to a third party.

A central challenge in application dialog replay is that the dialog intended for the original host will likely not be accepted by another without modification. For example, the dialog may include or rely on state specific to the original host such as its hostname, a known cookie, *etc*. In such cases, a straight-forward byte-by-byte replay to a different host with a different state (e.g., different hostname) than the original observed dialog participant will likely fail. These state-dependent protocol fields must be updated to reflect the different state of the different host for replay to succeed.

We formally define the replay problem. We present a solution which makes novel use of program verification techniques such as theorem proving and weakest pre-condition. By employing these techniques, we create the first *sound* solution to the replay problem: replay succeeds whenever our approach yields an answer. Previous techniques, though useful, are based on unsound heuristics. We implement a prototype of our techniques called *Replayer*, which we use to demonstrate the viability of our approach.

Categories and Subject Descriptors

C.2 [**Computer Systems Organization**]: COMPUTER COMMUNICATION NETWORKS

*This material is based upon work supported by the National Science Foundation under Grant No. 0448452. Jason Franklin performed this research while on appointment as a U.S. Department of Homeland Security (DHS) Fellow under the DHS Scholarship and Fellowship Program, a program administered by the Oak Ridge Institute for Science and Education (ORISE) for DHS through an interagency agreement with the U.S Department of Energy (DOE). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DHS, DOE, ORISE, or the National Science Foundation.

CCS'06, October 30–November 3, 2006, Alexandria, Virginia, USA.

General Terms

Security

Keywords

application protocol replay, weakest pre-condition

1. INTRODUCTION

In many scenarios, it would be extremely useful to automatically *replay* an application dialog as seen by one of the participants. This problem is termed the *replay of application dialog* [8]. Scenarios which benefit from automatic application dialog replay include dynamically analyzing programs through repeated execution in unique environments, demonstrating observed software vulnerabilities to interested parties, forensics, and determining the range of software versions vulnerable to an exploit.

However, as shown in [8], automatic replay of an application dialog is a challenging task. A primary issue is that a successful dialog may rely on state specific to the participants, such as hostnames, shared cookies, *etc*. These state-specific protocol fields cause problems when replaying the dialog to a different host. Cui et. al. recently proposed using machine learning to identify certain fields such as IP addresses, host names, etc., and then modify them accordingly to replay the application dialog [8]. They show that their approach works for many interesting protocols. However, this approach is intrinsically heuristics-based and cannot guarantee the correctness of the replay.

In this paper, we first develop a formal definition for the replay problem. The formal definition provides new insight into the problem, and opens the door to using formal techniques to solve the problem instead of relying on heuristics.

Based on our formal definition, we design an approach to solve the application replay problem by making novel use of existing program verification techniques. At a high level, we first build a *symbolic formula* of how the original host processed the application dialog. We then use an off-the-shelf decision procedure to derive an input tailored to a different host from the symbolic formula. This approach provides a *sound* solution to the application protocol replay problem. Unlike previous, heuristic-based approaches, our approach is general, and can handle unanticipated types of state-dependent protocol fields.

For example, successfully replaying a dialog may require modifying parts of the application dialog specific to the original host. However, changing the trace may result in an inconsistent dialog, e.g., check-sums on protocol messages may be incorrect. Thus, automatically determining which parts of a dialog to change, automatically changing them, and subsequently making the messages consistent in an automated fashion is difficult. Our techniques can soundly handle such cases, while previous work required manually applying domain-specific knowledge.

We describe and evaluate our implementation of *Replayer*, a working prototype of our approach for automatic protocol replay. We use Replayer to solve several examples of the automatic protocol replay problem.

Specifically, this paper makes the following contributions:

- We provide the first formal definition of the general replay problem.
- We show how to make novel use of the concept of weakest pre-condition and theorem proving to solve the replay problem. Our adaptation of these techniques results in the first *sound* solution to the replay problem.
- Our approach is general and enables successful replay in cases that the previous approach could not handle. For example, our replay can handle protocol messages which include check-sums.
- We have implemented our approach in a system called *Replayer*. Our experiments demonstrate the viability of this approach.

2. PROBLEM STATEMENT

Example 1 Program *P*

```
 1: session.cookie := counter
 2: counter := counter + 1
 3: send(session.cookie)
 4: recv(request)
 5: if (request.data.cookie != session.cookie) then
 6:     fail()
 7: else if (request.data.hostname != gethostname()) then
 8:     fail()
 9: else if (request.cksum != cksum(request.data)) then
10:     fail()
11: else
12:     process(request)
13: end if
```

Consider an application dialog between two hosts: an *initiator* and *host A* . The problem we address is how to *replay* the initiator's side of the network dialog to another party—*host B*. For non-trivial network protocols, it is insufficient to simply replay the exact input sent by the initiator. Certain protocol fields must be updated to reflect the state of host B for the replayed input to have the same effect on host B as it did on the host A.

Example 1 demonstrates three common types of protocol fields that must be updated to successfully replay a network dialog. The field checked on line 5 in the example is a cookie, which is an example of a *session*-specific protocol field. Cookies are opaque byte strings generated by one dialog participant, sent to the other, and then included in subsequent messages. They are used, for example, to identify a particular session or resource. Replaying a network dialog requires updating session-specific protocol fields to match the actual value that was sent by host B.

The field checked on line 7 of Example 1 is the host name of host B. This is an example of a *configuration*-specific protocol field. This type of protocol field is normally filled in using knowledge obtained by some out-of-band mechanism. Other examples of such state are user credentials, names of shared resources, etc. Replaying a network dialog requires updating configuration-specific protocol fields to reflect the configuration of host B.

The field checked on line 9 of Example 1 is a checksum over the rest of the request data, including the cookie and host-name

Term	Definition
S	Set of possible process states
$s_a, s_b, s_o, s_v \in S$	Initial state of host A, host B, observer, and verifier
$\sigma_a, \sigma_b, \sigma_o, \sigma_v \in S$	Post-state of host A, host B, observer, and verifier
I	Set of possible process inputs
$i_a, i_b, i_o, i_v \in I$	Input sent to host A, host B, observer, and verifier
$P : S \times I$	Program executed on host A, host B, observer, and verifier
Q	Set of possible post-conditions
$q \in Q$	Post-condition that is being satisfied
$\Phi : S \to Q$	Function to define a post-condition

Table 1: Terminology

fields. This is an example of a *consistency* protocol field. This type of protocol field is normally filled in using knowledge of the protocol. Another example of this type of protocol field is a length field—that is one specifying the number of bytes of one or more other protocol fields. If replay requires modifying other parts of the message to reflect host B's state, the corresponding consistency fields must be updated.

We next provide a formal and general definition of the protocol replay problem, which will lead to a formal and general solution.

2.1 Formal Definition

We define the problem of application protocol replay as follows. Let S represent the space of possible process states, and I represent the space of possible network inputs. A process with state $s \in S$ and input $i \in I$, executing a deterministic [1] program $P : S \times I \to S$, will result in a post-state $\sigma \in S$. The state s includes the *program counter* (the next instruction to be executed in the program), values of processor registers and memory, the state of the file system, *etc*. The input i is data written into the process via the network [2]. For simplicity, the final state σ also specifies the output produced during the execution.

A process on host A is executing the program P. While in initial state s_a, the process receives input i_a, and after continuing execution of the program, reaches a later state $\sigma_a \in S$, for which some *post-condition* $q \in Q$ is true, where Q is the set of all post-conditions defined as $Q = \{B | B : S \to \{\mathrm{T}, \mathrm{F}\}\}$. That is, $q(\sigma_a) = \mathrm{T}$. Intuitively, the post-condition will be a function that is true if and only if the input has a "similar" effect on host B as it did on host A. We discuss how to set the post-condition later in this section.

We wish to provide an input to Host B, who is also running a process executing program P, such that it reaches a state σ_b that satisfies the post-condition. *I.e.*, such that $q(\sigma_b) = \mathrm{T}$. However, the process being run by host B will be in a different initial state s_b. As a result, the input i_a seen by host A may *not* cause the post-condition to be satisfied. That is, it may be that $q(P(s_b, i_a)) = \mathrm{F}$. In Example 1, the original input i_a will have incorrect values for the cookie and hostname protocol fields, assuming that state s_b specifies a different cookie and hostname. Depending on the exact post-condition q being used, i_a will likely not satisfy $q(P(s_b, i_a))$ as a result.

The protocol replay problem is to modify the previous input to

[1] In Section 6.5 we discuss how to deal with nondeterminism such as scheduling, other inputs, *etc.*

[2] We refer to i as being a single input for simplicity. However, i could refer to multiple network messages.

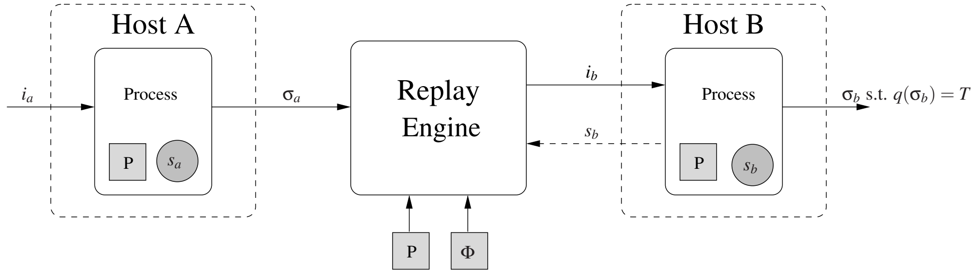

Figure 1: Our replay setting.

reach a σ_b *s.t.* $q(\sigma_b) = \mathrm{T}$. More specifically, the protocol replay problem is given input i_a, a post-condition q such that $q(\sigma_a) = \mathrm{T}$, and an initial state s_b, find a new input i_b such that $q(P(s_b, i_b)) = \mathrm{T}$.[3]

Setting the post-condition Intuitively, the post-condition q is true if and only if the replayed dialog has a "similar" result on host B as it did on host A. Exactly what is meant by "similar" will vary, depending on the purpose of performing application protocol replay.

In our approach, we define a function $\Phi : S \to Q$. That is, Φ takes the final state $\sigma_a \in S$ and returns the post-condition $q \in Q$. This formulation is illustrated in Figure 1. Φ, in turn, is dictated by the specific goal of the entity performing replay. Naively, one might specify Φ to produce a q that is satisfied if and only if the final state σ is the exact final state that was reached by host A, σ_a. However, since the state potentially encapsulates not only every memory value of the process, but the state of the machine it is running on, such a post condition is likely to be unsatisfiable.

A generic replay application may specify Φ to produce q that is true if and only if σ_b includes the same output as σ_a. This type of Φ would be suitable for many applications, as it would result in the same observable behavior on host B as was seen on host A.

More specialized applications may use a more specialized Φ. For example, one of the applications for protocol replay is to allow an exploit to be replayed to verify that a vulnerability exists and/or to further study the vulnerability. For this application, Φ might be "q returns T if and only if the security property that was violated in σ_a is also violated in σ_b," with a security violation such as "instruction x overwrites return address at memory address y."

3. OUR APPROACH

In this section we describe the design of our solution to the application protocol replay problem. Throughout the remainder of this paper, we refer to the host that received the original input as the observer, and the host to whom we are attempting to replay that input as the verifier. Our solution is illustrated in Figure 2. We first create a symbolic formula from the program P and the post-condition q. The resulting formula relates the input and the initial state to the final program state. We then substitute the verifier's initial state into the formula, and use a *decision procedure* to derive i_v for replay to the verifier.

3.1 Creating Symbolic Formulas for Application Replay

To replay an application dialog, we must determine what conditions are necessary for a host to accept the dialog and terminate in the a final state satisfying the post-condition. The program itself calculates a function from the initial state and input to a final state. This calculation can be expressed as a *symbolic formula* over the program state space and the input. We can further refine the symbolic formula to include only those final states that satisfy the post-condition. For example, the symbolic formula would include clauses that check for self-consistency, the relation between session-specific protocol fields, and relationships on the configuration state.

There are multiple approaches for computing a symbolic formula that represents a host accepting the replayed dialog. For example, one popular approach for this sort of problem is forward symbolic execution [16]. Forward symbolic execution entails "executing" the program on symbolic inputs, which results in a symbolic formula for that program path. If we symbolically execute all program paths, we could combine the formulas for each program path into a single formula for the entire program. We could then augment the symbolic formula for the program such that the formula is satisfied iff the program would accept the replayed dialog.

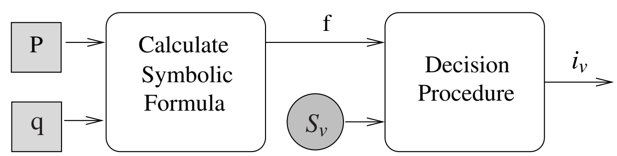

Figure 2: Our approach.

Our approach We take a different approach, where we compute the *weakest pre-condition* for a program P to terminate in a state satisfying post-condition q. A weakest pre-condition with respect to a post-condition q is a Boolean formula that characterizes the inputs and initial states which result in the program satisfying the post-condition. In our context, the weakest pre-condition formula characterizes the inputs and initial states that will cause the verifier to satisfy the desired post-condition. A satisfying assignment of values to variables in the weakest pre-condition produces an input that allows the verifier to reach the desired post-state.

The advantage of using the weakest pre-condition is that there are known methods for efficiently generating a single formula that considers multiple execution paths. In contrast, prior work employing forward symbolic execution generates a separate formula for each execution path.

More formally, the weakest pre-condition $wp(P,q)$ characterizes all inputs to the program P that execution will result in a terminating state satisfying q. At a high level, $wp(P,q)$ is the weakest pre-condition on i_v which implies executing P on i_v from initial state s_v will terminate in a state satisfying q.

Computing the Weakest Pre-Condition We compute the weakest pre-condition on P by:

[3]In the most general sense, we could also consider altering the program P or the host B's initial state s_b. While these approaches may be useful for some specialized applications, we do not address them in this work.

1. Translate P into the guarded command language (GCL). The GCL program, denoted P_g, is semantically equivilant to P, but much simpler for analysis.
2. Compute the weakest pre-condition $f = wp(P_g, q)$ in a syntax-directed fashion. The resulting formula f is a Boolean predicate over (verifier) program states and inputs.
3. Simplify the weakest pre-condition so it is more efficiently solved by the decision procedure in the next step.

Translating to GCL To calculate the weakest pre-condition for a program, we must first define how each instruction may affect program state. To simplify this task, we first translate the assembly instructions of P into a simplified language, called the *guarded command language* (GCL). The GCL has a relatively small number of distinct instructions, making it simpler to analyze. During this transformation, we also make all implicit modifications to the program state, such as processor status flags, explicit. The resulting program P_g is semantically equivalent to P, but can be reasoned about in a syntax-directed manner.

The GCL language constructs we use are shown in Table 2. Although GCL may look unimpressive, it is sufficiently expressive for reasoning about complex programs [9, 10, 13, 15] [4]. GCL is quite simple to understand. Statements in GCL mirror statements in assembly, e.g., store, load, assign, etc. Statements consist of a side-effect free rhs expression, and a lhs location to store the result. The lhs is always a variable name (i.e., a register) or memory location (both stack and heap locations are treated uniformly). $A;B$ denotes a sequence where statement A is executed, then statement B. $A \Box B$ is a choice statement where either A is executed or B, and corresponds loosely to a conditional jump statement. **assume** e assumes a particular (side-effect free) expression is true, and is used to reason about conditional jump predicates. **skip** is a semantic no-op, and provided to make subsequent analysis simpler. $lhs := e$ denotes an assignment of the expression e to location lhs. **true** and **false** are the logical constants (and we also allow for normal Boolean operators such as negation ($\neg$)).

GCL is best demonstrated with a simple example. The statement:

$$\text{if } (x < 0) \text{ then } x := x - 1 \text{ else } x := x + 1;$$

is translated as:

$$(\textbf{assume } x < 0; x := x - 1) \Box (\textbf{assume } \neg(x < 0); x := x + 1;)$$

System calls can be translated into a series of assignments to special variables. For example, input into the program via the `recv` system call can be written as a series of assignments to memory from $\texttt{input}_i$, for $i = 0$ to len, where len is the parameter passed to the system call specifying the maximum number of bytes read.

Likewise, data written into memory by other system calls, such as `gettimeofday`, reads from files or other sockets, *etc.*, can be represented as assignments to memory via specially named variables. The values of such variables can be considered part of the initial state s_v. We further discuss how to set these variables in Section 3.2.

Computing the weakest pre-condition We compute the weakest pre-condition for P_g in a syntax directed manner. The rules for computing the weakest pre-condition are shown in Table 2. Most rules are self-explanatory, e.g., to calculate the weakest pre-condition
$wp(A;B,Q)$, we calculate $wp(A, wp(B,Q))$. Similarly
$wp(\textbf{assume } e, Q) \equiv e \Rightarrow Q$.

[4] The GCL defines a few additional commands such as a **do-while** loop, which we do not use.

The one tricky rule is for assignment, i.e., calculating $wp(lhs := e, Q)$ where lhs is a variable name or memory reference and e is an expression. The rule $Q[lhs/e]$ specifies that all occurrences of lhs in the post-condition Q are substituted for e. If lhs is a variable, then we substitute all occurrences of lhs in Q for e. For example, $wp(j := i+1, j < 3) \equiv i + 1 < 3$. However we must take into account any possible memory aliasing relationships when lhs is a memory reference. Consider computing $wp(mem[w] := e, mem[t] < 3)$. If $t = w$, then the resulting weakest pre-condition is $e < 3$. If $t \neq w$, then this statement has no effect and the weakest pre-condition is $mem[t] < 3$. Therefore, the weakest pre-condition for $wp(mem[w] := e, mem[t] < 3)$ is

$$(\text{if } w = t \text{ then } e \text{ else } mem[t]) < 3$$

The complete weakest pre-condition calculation for post-condition $x < 3$ for our previous example is:

$$\begin{aligned} & wp((\textbf{assume } x < 0; x := x - 1) \Box (\textbf{assume } \neg(x < 0); x := x + 1;), \\ & \quad x < 3) \\ & \equiv (wp(\textbf{assume } x < 0; x := x - 1, x < 3) \\ & \quad \wedge wp(\textbf{assume } \neg(x < 0); x := x + 1, x < 3)) \\ & \equiv (x < 0 \Rightarrow x - 1 < 3) \wedge (\neg(x < 0) \Rightarrow x + 1 < 3) \end{aligned}$$

3.2 Obtaining the verifier's initial state

To replay an input to the verifier, we must substitute the initial state s_v into the symbolic formula before we can use it to find a satisfying input. There are several ways this state may be obtained and represented in the symbolic formula, depending on the type of state to be provided, and the particular application scenario.

Example 2 GCL

```
1: session.cookie := counter;
2: counter := counter + 1;
3: SENT := session.cookie;
4: request := INPUT;
5: assume(request.data.cookie != session.cookie) ⇒ fail()
   □ assume(request.data.cookie = session.cookie) ⇒
6: (assume(request.data.hostname != HOSTNAME) ⇒ fail()
   □ assume(request.data.hostname = HOSTNAME) ⇒
7: (assume(request.cksum != cksum(request.data) ⇒ fail()
   □ assume(request.cksum = cksum(request.data)) ⇒
8: process(request))))
```

We use Example 2 as an illustrative example. Example 2 is the program from Example 1 translated into GCL. Notice that the `if` statements have been converted to `assume` statements and that the system calls corresponding to `send`, `recv`, and `gethostname` have been converted to assignments to or from the special variables `SENT`, `INPUT`, and `HOSTNAME`, respectively. This example is *without* computing the weakest pre-condition, for greater readability.

In this example there are two parts of the verifier's initial state that are needed to satisfy the `assume` statements, and therefore to produce a successful replay: the value of the cookie, and the value of the host name. We first show how the appropriate state can be obtained if we have *direct access* to the verifier. We then show how in many cases, including this one, the necessary state can be obtained even when direct access to the verifier is unavailable, using *a priori* knowledge, and knowledge inferred from the verifier's output.

3.2.1 Direct Access

A,B $\in$ GCL stmt	::= A;B	
	| **assume** *e* (*e* is an expression)	
	| *lhs* := *e* ($lhs \in$ VARS $S \times I$)	
	| A $\Box$ B	
	| **skip**	
	| **true** | **false**	

GCL stmt	wp(stmt, Q)
assume *e*	$e \Rightarrow$ Q
lhs := *e*	Q[*lhs*/*e*]
A; B	wp(A, wp(B,Q))
A $\Box$ B	wp(A, Q) $\wedge$ wp(B,Q)

Table 2: The guarded command language (left), along with the corresponding weakest precondition predicate transformer (right).

The most straight-forward method of obtaining the verifier's state is to access the memory, registers, and system configuration directly. In this Example 2, the state of memory in s_v includes the value of *session.cookie*, allowing the corresponding protocol field to be correctly updated.

In this example, the host name is derived during execution, from a system call. Hence, the initial values of memory and registers are insufficient to find this part of the verifier's state. However, we can use direct access to the verifier to predict what the system call will return, and provide that value as part of the initial state, as an assignment to $HOSTNAME$. For many system calls, including this one, the return value can be found simply by executing that system call in a separate process on the verifier.

3.2.2 Non-direct access

Obtaining the initial state is more challenging if direct access to the verifier is unavailable. However, in many cases the necessary state can be automatically obtained by analyzing previous output of the verifier, and using *a priori* knowledge. In Example 2, we can obtain the value of the cookie using output sent by the verifier, and we can obtain the host name using *a priori* knowledge.

Inference from output In Section 2, we described three types of protocol fields that must be updated for replay to be successful: session-specific fields, configuration-specific fields, and consistency fields. Session-specific fields can typically be updated using only state inferred from output of the verifier.

In Example 2, `session.cookie` is an example of such state. By inspection, clearly if we know the value sent by the verifier, `SENT`, we can infer the value of `session.cookie`. If we substitute the variable `SENT` with x, where x is the value actually sent, then the decision procedure will likewise be able to infer that `session.cookie` $= x$, and further that $\texttt{INPUT}_i = x$, where i is the offset of the cookie field.

A minor caveat to this approach is that $SENT$ will not appear directly in the weakest pre-condition. Therefore, to derive state from the output in this way, we must incorporate the weakest precondition of the corresponding output variables into the symbolic formula provided to the decision procedure.

A priori knowledge Configuration-specific protocol fields sometimes cannot be derived directly from the output of the verifier, as is the case with the `hostname` field in Example 2. In this example, a client must have *a priori* knowledge of the host name, or be able to find it by some out-of-band means.

We can handle some of these cases by modeling system calls that read configuration state, as we do with the call to `gethostname` in Example 2. When such a variable appears in the symbolic formula, we can attempt to find its value by some out-of-band means. In this case, we could perform a reverse DNS lookup of the remote host, and use that data to provide an assignment to `HOSTNAME` in our symbolic formula.

As in the previous case, we must augment the symbolic formula with the weakest pre-condition of any such system calls we want to handle, because they will not appear directly in the weakest precondition of the post-condition.

3.3 Finding a satisfying input

The output of the weakest-precondition phase is a boolean formula f over program states $s \in S$ and input variables $i \in I$. We then assign values to the state variables s from the verifier's initial state s_v, as discussed in the previous section. We then use an off-the-shelf *decision procedure* to provide an example assignment i satisfying the weakest pre-condition. This i will result in a post-state that satisfies the post-condition.

The decision procedure provides such an assignment if one exists given initial state s_v. Otherwise, the decision procedure returns that the formula f with the state variables specified by s_v is not satisfiable.

If the decision procedure returns an assignment, the input i_v can be directly constructed using the assignments to the input variables. Again, because we are using sound techniques, such an input *will* satisfy the post condition.

In most cases, we will not be able to incorporate every possible execution path into the symbolic formula. We further discuss how we bound the program before computing the weakest pre-condition in Section 4.2.2. If the decision procedure is not able to find a satisfying assignment to the formula, it may still be satisfiable via an execution path not included in the formula. If desired we could backtrack and build a bounded program that includes additional execution paths, and try again. Note again that because there may be *infinite* paths in the full program, it is possible that we will never find a satisfying input regardless of how many paths we add to the bounded program. This is unsurprising, since in the general case one could easily construct a program and post-condition where finding a satisfying input can be reduced to deciding the halting problem.

Finally, there are of course cases in which *no* input will satisfy the post-condition given the initial state. For example, the verifier could be configured to not accept any incoming connection.

4. IMPLEMENTATION

We have implemented a proof-of-concept of our approach in a tool called *Replayer*. We describe the relevant implementation details in this section.

4.1 Trace recorder

We use Valgrind [21] to produce the execution trace T. Valgrind is an open source dynamic binary rewriting tool. Our choice was made for convenience, many other tools also produce traces [5, 20, 25]. We wrote a Valgrind plug-in that produces a log of the first address of each basic block executed.

4.2 Symbolic Formula Generator

We have built an analyzer that reads in the binary program P and the execution trace T, and outputs the corresponding weakest pre-

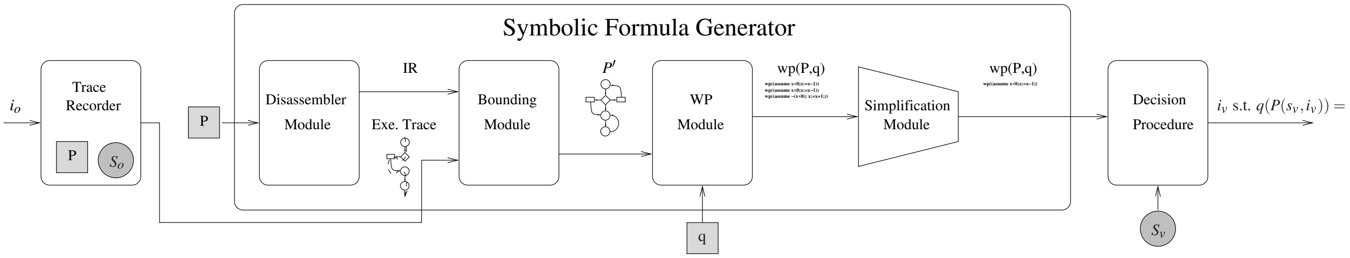

Figure 3: The Replayer design and implementation

condition formula. The code base for the analyzer is approximately 19,000 lines of C++ code. The code has several modules.

4.2.1 Disassembler module

The disassembler module is an extension of [6]. This module is responsible for disassembling the basic blocks logged in the trace and converting the assembly into an unambiguous intermediate representation. Translating to the intermediate representation, although straight-forward, requires us to explicitly model the effects of overflow, underflow, sign extension, etc. We simplified this task by first using Valgrind's libVEX to translate the assembly instructions to VEX—Valgrind's intermediate representation, thus saving us from having to specify every IA-32 instruction. However, since VEX is designed to generate efficient executable code rather than easily analyzable code, we transformed VEX to our own simpler intermediate representation. In particular, processor status flags in VEX are updated at run-time just before they are accessed. In our IR, status flags are instead updated explicitly.

One additional problem we must deal with is memory reads and writes, some of which may be unaligned. We call a memory read unaligned if it doesn't correspond to an atomic write. For example, consider a 32-bit write to memory locations 1-4. A subsequent read of byte 2 is unaligned. To address this, we post-process the assembly so all writes and reads are single bytes, e.g., a 32-bit write becomes 4 8-bit writes.

4.2.2 Bounding module

There are a large number of possible execution paths through most programs; infinitely many if the program does not halt. To reason about a program at all, we must convert the program to one that has a finite number of execution paths. A common technique for doing this is to bound the number of times each loop in the program executes, while adding an additional check to ensure that executions that would have executed a loop a greater number of times are not considered. Any reasoning over a program transformed in this way is still *sound*, but may not be *complete* since not all possible execution paths are considered.

We employ this technique, transforming the original program P into a modified program P'. Further, we use an execution trace T of the execution path followed by the observer as it processed the original input i_o to help determine *how* to bound the program.

We already know that a process executing P, with initial state s_o will reach a post-state that satisfies the post-condition (by definition), following the execution path specified in T. Intuitively, it is likely that the program can reach a post-state satisfying the post-condition by following an identical or similar execution path, even when starting in a different initial state s_v. Therefore, it makes sense to bound the program to execution paths similar to that in T. We call this *trace-guided bounding*.

In our implementation, trace-guided bounding bounds the program to follow the *exact* execution path from the trace T. This results in the most scalable, but the least complete solution. Therefore, it may be desirable to also consider *similar* execution paths. For example, one might consider allowing loops to be executed n greater or fewer times than they were executed in the trace T. One might also consider executions where alternate paths of "diamond" structures in the control flow graph are executed. For example, by inspection it makes little difference whether the `if` or the `else` clause is executed in `if (input > 3) printf(''foo''); else printf(''bar'');`.

In our current implementation, we bound the program to follow the exact path specified in T. We build a program consisting of the concatenation of each instruction executed in the trace T. All conditional and indirect jumps are replaced with direct jumps to the destination they jumped to in the trace T. To ensure soundness, we later add the corresponding condition from each jump (or calculation in the case of an indirect jump) to the post-condition, thus ensuring that we only consider inputs that would actually result in this execution path.

4.2.3 Weakest pre-condition module

The weakest pre-condition module translates the intermediate representation into the guarded command language from Table 2. The weakest pre-condition module implements the predicate transformers shown in Table 2. This code takes as input the GCL and an arbitrary post-condition, and outputs the corresponding weakest pre-condition formula.

4.2.4 Simplification module

After computing the weakest precondition, the model consists of a single, relatively large, formula. We perform a number of simplifications on this formula to reduce the size of the model. Smaller models are easier for the decision procedure (described next) to reason about.

We perform:

- Arithmetic simplification and re-association. For example, the expression $4 + esp + 4 < e$ would be simplified to $8 + esp < e$. Arithmetic simplification and re-association leaves arithmetic expressions as a sum-of-products.
- Boolean simplification. For example, many memory accesses are a constant offset from the stack pointer—`esp` on the IA-32 architecture. As a result, the conditions in many of the `if-then-else` clauses that test for memory aliasing are of the form $esp + x = esp + y$, where x and y are constants. Obviously, such a condition is true if and only if $x = y$, allowing us to simplify away the `if-then-else`.
- Common sub-expression elimination. For example, consider the formula $a + 2 < 3 \wedge a + 2 < 3$. We eliminate common sub-expressions via a `let` binding, e.g., `let` $t = a + 2 < 3$ `in` $t \wedge t$, which results in $a + 2 < 3$ being considered only

once by the decision procedure. Note that applying Boolean simplification will further reduce the formula to `let` $t = a+2<3$ `in` t.

We have found these simple optimizations to be extremely important. In many test cases simplification reduced time spent in the decision procedure by orders of magnitude. Our running example is simplified as:

$$(x<0 \Rightarrow x<4) \wedge (\neg(x<0) \Rightarrow x<2)$$

Note that the above formula could be further simplified, but we leave such simplification as future work.

4.3 Replay Engine

State Extractor For simplicity, we use the direct-access method of finding the verifier's state, as described in Section 3.2. The state extractor uses ptrace to read the state s_v of memory and registers of the process running program P at the point where P is waiting for input (*e.g.* blocked on a `read` system call).

The Replayer module substitutes the state s_v into the weakest pre-condition $wp(P,q)$, and runs the simplification module again. It then translates the weakest pre-condition into the appropriate syntax for a decision procedure.

Decision Procedure The decision procedure is a modular component, and we could in theory use any off-the-shelf product. We currently use STP [14], a decision procedure that specializes in modeling bit-vectors. After translation, the module asks the decision procedure for a satisfying assignment of values to input variables. The decision procedure will either output a satisfying input i_v for state s_v, or output that there does not exist such an input.

When a satisfying input is found, we replay the new input i_v and verify that the post-condition is satisfied. This step serves as a self-check of our implementation.

5. EVALUATION

We evaluate Replayer on several variations of Example 1. Each test is compiled as an unoptimized C program. We specify Φ as "execution reaches process(request)". In each test, we take the initial state at the point of the *recv* at line 4. We take the final state at the point where `process(request)` is called at line 12. Our measurements are performed on a machine running Ubuntu Linux version 5, with a Pentium 4 2.20 GHz CPU, and 1 GB of RAM.

When measuring performance, we consider two steps. The first step is to build and simplify the symbolic formula. This step needs to be performed only once to replay a particular observed input i_o. The model can then be used to replay the input to any number of verifiers. The second step is to substitute the initial state of a particular verifier into the model, performing any additional simplification of the model, and use the decision procedure to find a satisfying input. The second step must be performed each time the input is replayed to a verifier with a different initial state. We also provide the number of instructions in the executable. Note that the size of the trace can exceed the number of instructions as the instruction trace may include a single instruction multiple times, e.g., for when a loop is executed.

Simplification effectiveness Naturally, if we want to replay an input more than once, it pays to pre-process the symbolic formula as much as possible, to minimize the performance overhead of solving the symbolic formula. We evaluate the performance for several strategies, beginning with the naive approach of converting the entire GCL program corresponding to the trace to the language of the theorem prover, instruction by instruction. We show that performance is greatly improved by instead computing the weakest pre-condition, and is further improved by simplifying the weakest pre-condition formula.

We consider a version of Example 1 with only the test for the cookie field enabled. Here, the cookie field is a four-byte integer. Keep in mind that while this example is very simple at the source code level, the object code is significantly more complex.

In Table 3, we compare our performance when converting the entire GCL version of the program into the decision procedure's language, when instead computing the weakest pre-condition over the GCL program, and when performing our additional simplifications over the weakest pre-condition formula.

As expected, the formula resulting from computing the weakest pre-condition is much more efficiently solved by the decision procedure than the formula resulting from the full GCL program. The weakest pre-condition formula takes roughly one third the time for the decision procedure to solve.

Interestingly, our simplification module improves the performance of solving the final formula by two orders of magnitude. Further, the time to *generate* the formula in the first place is greatly reduced. This is because we perform simplification on the intermediate forms of the formula as we calculate the weakest pre-condition, reducing the complexity of computing the weakest pre-condition itself. Some of these simplification techniques could be built into decision procedures. [5]

Updating a checksum We next enabled the check for the checksum, in addition to the check for the cookie. Hence, because the replayer must update the cookie, it must also update the checksum. In our program, the checksum is the integer addition of the data. We evaluate performance using a 16 byte checksum (4 integer additions), and an 80 byte checksum (20 integer additions). An advantage of our approach over machine-learning based approaches such as Roleplayer [8] is that we can replay protocols that have such relationships, without needing to know the exact algorithm ahead of time.

Table 4 shows the execution time for each Replayer step for when only the cookie check is enabled (same as in Table 3), for when the checksum is computed over 16 bytes of input, and for when the checksum is computed over 80 bytes of input. As one would expect, the checksum computation significantly increases the complexity of the generated formula. Replayer handles each of these cases in a reasonable amount of time, though further work will be needed to scale Replayer to larger formulas.

We have demonstrated the validity of our approach and of our implementation prototype. While further work is necessary for our approach to scale, we believe that our initial results are promising. We discuss some possible strategies to improve the scalability of our approach in Section 6.6.

6. DISCUSSION

6.1 Preserving similarity to the original input

Replayer is designed to soundly address the problem formulation given in Section 2. That is, Replayer produces an input i_v that will replay and reach the desired final state satisfying the desired post-condition. Whether or not the produced i_v resembles the original input i_o depends completely on the post-condition. Here, we discuss how the post-condition can be specified so that i_o and i_v are textually similar.

[5] Shortly after this writing, the authors of the STP decision procedure did implement additional simplifications inside STP based on our feedback, greatly improving its performance in these experiments.

	Building the Formula (s)	Total time to solve formula (s)
Full GCL program	.944	34.4
WP without simplification	12.2	11.3
WP with simplification	1.15	0.142

Table 3: Performance improvements from weakest pre-condition and simplifications

	Cookie only	Cookie + 16 byte checksum	Cookie + 80 byte checksum
Executable size (w and w/o glibc)	87188/329	87188/329	87188/329
IA-32 instructions in trace	35	84	229
1. Trace to GCL (s)	.901	.947	.892
2. Compute WP (s)	.205	7.92	355
Total time to build formula (1+2) (s)	1.106	8.867	356
3. Substitute and resimplify (s)	.029	.652	7.54
4. Translate to decision procedure (s)	.013	.142	2.02
5. Compute decision procedure (STP) (s)	0.10	.95	5.73
Total time to solve formula (3+4+5) (s)	.142	1.744	15.29

Table 4: Checksum replay performance.

For example, suppose that the original input i_o is a message to an SMTP server that causes the SMTP server to send an email message. Suppose the post-condition only specifies that the input should trigger a message on the verifier host. Using our current approach, we would be likely to generate an i_v that *also* causes an SMTP server to send an email message, but the body of that message would likely be gibberish, rather than the contents of the message in i_o. In general, specifying a Φ that, for messages in any possible network protocol, generates a q that specifies that i_v is entirely semantically equivalent to i_o is impossible.

There are several techniques for creating a post-condition such that i_o and i_v are textually similar. Each technique *constraints* the input variables for non-state-dependent protocol fields to be the value that they took on in i_o, thus only allowing the decision procedure to select new values for the state-dependent protocol fields.

The challenge to accomplish this is that we would need to *identify* which parts of the input correspond to state-dependent protocol fields. There are several techniques we could use to do this. Note that *whichever* bytes of the input we choose to constrain, the system is still sound. However, every additional constraint risks causing the symbolic formula to become unsatisfiable. In particular, if we mistakenly constrain a state-dependent protocol field to its original value, of course we will be unable to find a satisfying input.

There are several techniques we can use to heuristically identify which bytes of the input do not correspond to state-dependent fields, and may hence be safely constrained. First, we can of course constrain any input bytes that do not appear at all in our symbolic formula, without reducing the completeness of the symbolic formula. Unfortunately, such bytes will not exist in most protocols. For example, in a text-based protocol, even input bytes that are otherwise ignored by the program will likely be constrained by the symbolic formula to not be `NULL`. Hence, this technique of identifying constrained bytes is not likely to be useful in practice.

A more promising technique is to identify the state-dependent fields, and consistency fields, and then constrain the rest of the input. For example:

- **Session-specific fields**, such as cookies, are characterized by the program sending data derived from some internal state (*e.g.*, a saved cookie), and later comparing that internal state to subsequent input.
- **Configuration-specific fields**, such as the name of a host, may be compared with data derived from another system call— *e.g.*, reading a configuration file. Another possibility is that the input data is passed as a *parameter* to a system call— *e.g.* to open a specified file.
- **Consistency fields**, such as a check-sum, result in an expression in the symbolic formula comparing a relatively small protocol field, *e.g.* a check-sum, with an expression involving a relatively large protocol field, *e.g.*, the rest of the request.

Another technique is to use an iterative greedy algorithm to determine which parts of the input can remain the same as i_o. First, we would use the decision procedure to find a satisfying input i_v as before. We would then attempt to constrain one of the input variables to have the same value as in i_o, and try again. If the decision procedure succeeds, we can continue to constrain that variable to have the same value as in i_o, otherwise we un-constrain it again. We can continue this process until no more bytes can be constrained without causing the decision procedure to fail.

The greedy part of this approach stems from the order to attempt to constrain the input bytes. Suppose the message has a consistency field such as a check-sum. While it may be possible to keep the consistency field value from the original input and allow the decision procedure to find new values for the data fields, it would be preferable to keep the data field values from the original input and derive a new consistent value for the consistency field. To encourage this property, we greedily prefer to constrain bytes that appears on the other side of a comparison operator as the fewest number of other input variables in the Boolean formula. Thus if we have $cksum = a+b+c+d$, we would prefer to constrain a through d before $cksum$.

We leave implementing and evaluating these techniques as future work. If successful, these techniques could greatly increase Replayer's ability to act as a generic protocol replay tool, by helping preserve any hidden semantics of the replayed message.

6.2 Generating an observer-independent protocol replay engine

In our current solution, we use the post-state σ_o and the execution trace T obtained by monitoring the *observer* as it processes the original input i_o. In some replay applications, this information may not be available. In particular, it would be useful to replay an input obtained from a logged network trace.

We may be able to build a replay system that can replay messages of a particular protocol, after observing and building symbolic formulas for most of the distinct types of messages in that protocol. Assuming we are able to identify which fields are state-dependent using the techniques described in Section 6.1, we could constrain the non-state-dependent variables to the values of some other input i_2 to replay *that* input. To build a tool to replay any message of a particular protocol in this way, we would also need some mechanism to determine *which* of our symbolic formulas to employ to replay a particular input. One way of doing this is to build a *signature* of each type of message when generating the initial symbolic formulas. We can build these signatures using similar techniques to those described in [6]. That is, given the original observed σ_o and input i_o, we determine which parts of the input must remain the same to satisfy the formula. The bytes that cannot change will typically correspond to protocol keywords, which identify the type of message being sent. These protocol keywords can be built into a signature, which can later be used to identify subsequent inputs of the same message type.

6.3 Different program versions

In our current design, we assume that observer and the verifier are running the exact same program, P. In practice, we may be able to replay to a verifier that is running a different program P', if P' behaves as P in externally observable ways. This will often be the case for two slightly different versions of the same program, or perhaps even for two independently developed programs implementing the same protocol.

Direct access to the memory and registers of the verifier process are unlikely to be useful in supplying the initial state s_0, because P' will have a different memory and register allocation. However, if the necessary state can be inferred from the verifier's output and from a priori knowledge as described in Section 3.2, and if the program P' implements the same protocol specification as P, an input i_v derived from our symbolic formula is likely to result in the same *externally observable* behavior on the verifier as i_o did on the observer.

6.4 Complexity of finding a satisfying input

The general problem of finding an input that satisfies an arbitrary post-condition can easily be shown to be undecidable. *E.g.* the post-condition could be "Program outputs 1 iff f(s_v, i_v) produces a program that halts." Naturally, we do not claim to be able to solve the problem for such post-conditions. In practice, we expect most useful post-conditions to be relatively simple, such as the examples given in Section 2.

Likewise, even a simple post-condition such as "execution reaches the same final *eip* as specified in σ_o" could be thwarted by a program that, for example, checks whether a cryptographic pseudo-random function computed on the input i_v is equal to the state s_v. However, for most programs, one would not expect the problem of finding an input to cause that program to reach a desired final state to intentionally be made into a hard problem. With few exceptions, programs are designed so that inputs can easily be constructed to cause a program to reach a desired state.

Note that handling common cases of cryptographic functions is not fundamentally difficult. For example, suppose that a protocol field of the input must include a correct cryptographic message authentication code (MAC) of the rest of the input. There is no need to "invert" the MAC to derive a correct input. The cryptographic key is part of the process state (or the process would not be able to verify the MAC itself). Hence, it is possible to derive a correct input by first satisfying other constraints on the input, and then performing a forward execution of the MAC computation code, which again is part of the program itself. We do not provide a general solution to automatically recognize and handle such cases in this work, though it would be straight-forward to include logic to recognize and handle calls into common cryptographic libraries. We believe a more general solution may be possible, which we leave for future work.

6.5 Non-determinism

Our approach assumes that the program behaves in a deterministic manner between the time the input i_v is read in, and the time that the final state s_v is reached. This is the case in many programs. In the presence of non-determinism not accounted for in our symbolic formula, our approach is no longer sound. However, it is likely that in many cases it would still work, as intuitively programs are constructed to *behave* in a deterministic manner based on their input, even when some non-determinism is present.

For example, consider a program that uses threads that are preemptively scheduled by the kernel. From the process's perspective, the scheduling of these threads is non-deterministic. However, assuming the absence of race conditions[6], the particular scheduling order of the threads has no effect on the final state reached by the program. Hence, we can compute the weakest pre-condition based on the scheduling order we observed in the original execution trace, and assume that the weakest pre-condition holds for other scheduling orders. The same reasoning applies to a program that is concurrently processing other requests, either via threads or via asynchronous I/O.

A potential challenge exists when the weakest pre-condition depends on the result of a system call made in between reading the input and reaching the post-condition. For example, the behavior of the program may depend on data read from a file on disk after receiving the replayed input. As we showed in Section 3.2, some system calls can be incorporated into our symbolic formula as part of the verifier's initial state. That is, if we can predict what data will be returned by a particular system call, we can substitute that data into the symbolic formula, thus removing the non-determinism. Hence, approach is still sound in these cases if we can accurately determine what these system calls will return. As discussed in Section 3.2, whether we can do this depends on the type of system call, and how much access we have to the verifier.

6.6 Insights into improving scalability, performance, and efficiency

Our approach was fundamentally motivated by the need for sound techniques for application replay. Our experience indicates that implementation details can make huge differences in efficiency, scalability, and performance. In particular, the availability of a compiler optimization will significantly (often exponentially) result in smaller and easier to prove formulas. We have found ourselves repeatedly implementing common compiler optimizations to get better performance. For example, without our simple algebraic simplifications, the 16 byte check-sum took over 80 minutes to compute the weakest pre-condition, instead of the current 7.9 seconds.

We believe further dramatic improvements are easily obtainable by implementing known compiler techniques. For example:

[6]We do not attempt to address the problem of reproducing race conditions in this work. However, it may be possible to extend our approach to do so, for example by computing the weakest pre-condition over the scheduling algorithm as well.

- We lack alias analysis, which often results in formulas (sometimes exponentially) larger than needed. During execution, many instructions are memory references, either to the stack for local variables spilled due to register contention, or to the heap. We take a strictly sound approach, where any two variables that may be aliased are considered aliases, resulting in a `if-then-else` formula as described in Section 3.1. The `if-then-else` can potentially double the size of the program. Implementing x86 alias analysis such as value-set analysis [4] would significantly help.

- Further simplifications (e.g., common sub-expression elimination, more aggressive simplification, global value numbering, etc.). As mentioned in Section 4.2.4, we found that such simplifications can result in an order of magnitude speedup.

- We use a classical approach to calculating the weakest precondition. Flanagan and Saxe [13] propose a method that can exponentially reduce formula size.

7. RELATED WORK

The work closest to our own is that of Cui *et. al.* [8]. Cui *et. al.*develop a heuristic-based approach to automatically identify and update application fields in protocol dialog. A primary aim of their work is to decouple application semantics from the replay process. This decoupling requires the identification of application protocol dependent fields which must be modified for correct replay. The process by which different classes of fields are identified, modified, and replayed is a manual process based on the semantics of each field type. The accuracy of the developed techniques is not guaranteed and new heuristics must be developed to handle new classes of dynamic fields.

Several projects have addressed the problem of network traffic replay. They typically focus on the network or transport layers [1,2,7] or include application semantics through the manual development of application-level responders or plug-ins [22,23,27]. Our approach works at the application layer and utilizes the application itself for semantic information rather than a simplified version of the application encapsulated in a plug-in.

Replaying the execution of a program has been the focus of numerous projects [3,11,17,18,24,26]. These projects typically focus on ensuring deterministic execution in the face of non-determinism or logging the precise sequence of instructions for debugging, intrusion analysis, or simple instruction-by-instruction replay. Most of these previous approaches simply log the non-deterministic event, for example, the order of memory accesses on a multiprocessor, or log the individual instructions executed. However, this is insufficient to correctly replay the execution of a program on a machine with different state. We incorporate the application and the machine state in order to soundly determine the input that would arrive at the same post condition.

We compute the weakest pre-condition using direct, classical techniques. More advanced techniques can be implemented and may resulting significant improvements. For instance, Flanagan and Saxe which can significantly reduce the size of the weakest pre-condition formula [13]. We adopt the standard technique of unrolling loops, which may lead to incompleteness, however, unrolling is not necessary if loop invariants can be provided. Others have explored automatically determining loop invariants, e.g., [12, 19].

8. CONCLUSION

We developed a solution to the protocol replay problem: that of replaying an application dialog observed by one host to another host. The challenge in protocol replay is that a dialog sent to one host will likely not be accepted by another. A simple byte-by-byte replay to a host that is in a different initial state than the original will fail. We developed a general and formal definition of the replay problem. The solution developed in this paper makes novel use of programming language techniques such as theorem proving and weakest pre-condition. By apply these techniques, we developed the first *sound* solution to the protocol replay problem. We implemented and evaluated a prototype of our replay system called *Replayer*. Our evaluation demonstrates both the viability and generality of our problem formulation and corresponding approach.

Acknowledgments

We would especially like to thank Cristian Cadar, David Dill, and Vijay Ganesh for their generous help with CVCL and STP. We would also like to thank Ivan Jager, Eric Li, Vern Paxson, and the anonymous reviewers for their helpful comments and suggestions during the preparation of this paper.

9. REFERENCES

[1] Cybertrace. http://www.cybertrace.com/ctids.html.
[2] Tcpreplay: Pcap editing and replay tools for *NIX. http://tcpreplay.sourceforge.net.
[3] David F. Bacon and Seth Copen Goldstein. Hardware-assisted replay of multiprocessor programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1991.
[4] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proc. Int. Conf. on Compiler Construction*, 2004.
[5] P Bosch, A Carloganu, and D Etiemble. Complete x86 instruction trace generation from hardware bus collect. In *23rd IEEE EUROMICRO Conference*, 1997.
[6] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2006.
[7] Yu-Chung Cheng, Urs Hoelzle, Neal Cardwell, Stefan Savage, and Geoffrey M. Voelker. Monkey see, monkey do: A tool for tcp tracing and replaying. In *Proceedings of the 2004 USENIX Annual Technical Conference*, June 2004.
[8] Weidong Cui, Vern Paxson, Nicholas C. Weaver, and Randy H. Katz. Protocol-independent adaptive replay of application dialog. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium*, February 2006.
[9] D.L. Detlefs, K. Rustan M. Leino, G. Nelson, and J.B. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center, December 1998.
[10] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.
[11] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.
[12] M. D. Ernest, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evoluation. *IEEE Transactions on Software Engineering*, 27(2), Feb 2001.
[13] C. Flanagan and J.B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the 28th ACM Symposium on the Principles of Programming Languages (POPL)*, 2001.
[14] Vijay Ganesh and David L. Dill. System description of STP. `http://www.csl.sri.com/users/demoura/smt-comp/descriptions/stp.ps`, August 2006.

[15] David Gries, editor. *Programming in the 1990's: An Introduction to the calculation of programs*. Springer Verlag, 1990.

[16] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), July 1976.

[17] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 USENIX Annual Technical Conference*, April 2005.

[18] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 36(4):471–482, 1987.

[19] K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In *Asian Symposium on Programming Languages and Systems APLAS*, 2005.

[20] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of 2005 Programming Language Design and Implementation (PLDI) conference*, june 2005.

[21] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In *Proceedings of the Third Workshop on Runtime Verification (RV'03)*, Boulder, Colorado, USA, July 2003.

[22] R. Pang, V. Yegneswaran, P. Barford, V. Paxson, and L. Peterson. Characteristics of internet background radiation. In *Proceedings of Internet Measurement Conference*, October 2004.

[23] Niels Provos. A virtual honeypot framework. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[24] M. Russinovich and B. Cagswell. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the 1996 Conference on Programming Language Design and Implementation*, May 1996.

[25] P. A. Sandon, Y.C. Liao, T.E. Cook, D.M. Schultz, and P Martin de Nicolas. Nstrace: A bus-driven instruction trace tool for powerpc microprocessors. *IBM Journal of Research and Development*, 41(3), 1997.

[26] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou. Flashback: A light-weight rollback and deterministic replay extension for software debugging. In *Proceedings of the 2004 USENIX Annual Technical Conference*, June 2004.

[27] A. Turner. Flowreplay design notes. http://www.synfin.net/papers/flowreplay.pdf.