Exploit-Robust Implementation of Protocols Through Formal Method Driven Unambiguous and Attack-Intelligent Specifications

Prabhaker Mateti

Abstract

A large number of protocol related vulnerabilities occur because the corresponding implementation failed to anticipate packets that do not correspond to the protocol specification. Presently, most of the protocol specifications define what constitute "right" packets relevant to the protocol and they specify what the response should be for such packets. They often remain silent on what the protocol implementation should do for packets which deviate from the specification.

Protocol RFCs continue to be written as above, and the corresponding implementations continue to be vulnerable. Attack techniques are becoming more sophisticated in exploiting non-robust protocol implementations. There are not many solutions available to the protocol implementation community to counter these attack techniques.

We propose a solution to the above problem. First, we fix the specification in such a way that there is no ambiguity for the implementer in those areas where past exploit techniques exist and future exploits can be predicted. Secondly, we classify the "bad" packets in the state-space through formal description techniques. Third, at certain locations in the source code implementation of the protocol, we insert assertions based on this classification. If the platform on which the protocol implementation is running is powerful enough, we can enable the assertions as run-time checks.

Contents

1 Introduction

Sending cleverly crafted packets to a protocol engine so that it is impaired has become a common attack technique. A large number of such protocol related vulnerabilities occur because the corresponding implementation failed to anticipate packets that do not correspond to the protocol specification.

1.1 Problem Description

While it is desirable to have security savvy protocol implementers and RFC writers in the long run, it is more realistic that we expect neither of them to be aware of various attack techniques and to write exploit-robust code. Typical software engineering principles suggest that an implementer should faithfully follow the given specification and produce the desired output only when the input satisfies specified input conditions. He is not expected to assume or deal with conditions and inputs which do not conform to the given specification. Should that become necessary, one should go back to the requirements analysis phase and update the requirements and specifications.

As a result, we expect the protocol implementations to continue to be vulnerable as long as protocol specifications are written ambiguously, and written without considering common attack techniques.

1.2 A Few Example Incidents

A network device rebooting is not an uncommon sight when a packet with large sized data or a zero length data is sent to it. It is not uncommon either to observe memory leaks on a device because the corresponding protocol stack did not implement timeouts for various connection stages. Further examples are given in Sections 5 and 7.

2 Causes Of The Problem

2.1 Protocol Design Problems

A protocol design itself may be exploitable. TCP sniper attacks, ARP poisoning, DNS attacks are examples of these. The primary cause for this are the security-naive protocol RFC writers.

2.2 Inadequate Protocol RFCs

Protocol definitions are written up as technical reports in English. This is prone to ambiguities and misinterpretation. It is a well-known but perhaps unacknowledged fact that different network protocol stacks would not be interoperable had it not been for the wide accessibility of source code that implemented the early and foundational protocols such as TCP. Even today, one can safely predict that, if an implementation team is given only the RFCs and denied all access to source code of other implementations, the result would not be interoperable.

2.3 Silence in Specifications

Presently, most of the protocol specifications define what constitute "right" packets relevant to the protocol and they specify what the response should be for such packets. They remain silent on what the protocol implementation should do for packets (meaning header values) which deviate from the specification.

An implementer takes the "silence" in the specification as "design freedom", but usually they have not factored in the security issues. Some ambiguity also exists because of the "language" used by the specifications, namely English.

The protocol implementer also usually does not have knowledge of security issues, past exploits, common attack techniques that impact his module and hence the overall stack and device.

2.4 Attack-Technique Ignorant Protocol Implementations

The present protocol specifications also do not specify what the implementation should do with respect to common attacks related to packets, for example flooding with packets, and not responding with an appropriate "reply", or deliberately delaying the reply.

3 Goals

The over all goal is to bring security concerns into specifications, and implementations of protocols.

3.1 Problem Definition

Our primary goal is to solve the following problem. We are given the RFC of a protocol. We are given the source code of an implementation of a protocol. We assume that the implementation is correct with respect to the protocol. We are also given reports of past exploits of the protocol and the implementation. We are asked to reduce the security risk posed by the protocol and its implementation. We assume that rejection of either the protocol or its present implementation is not an option. The space of acceptable solutions includes enhancing the protocol with minimal incompatibilities, and making the code more robust by careful revision of code. We assume that it is permissible to have a range of solutions that depend on issues such as the computing power of the platform.

3.2 Improving the RFCs

The long term solution to robust implementations lies in writing much better RFCs. Improvements in eliminating ambiguity, and identifying devious classes of inputs are possible, and this task is an explicit goal of ours.

3.3 A Best Practices Example of an RFC

We would like to influence future RFCs, and encourage others to write clarifications of existing RFCs. This is best done through an example best practices document.

3.4 Explicit Description of Design Freedoms

A specification document ought to permit as much design freedom as possible. When a specification is silent on an item, the interpretation in the past has been that the particular item is irrelevant to the spec and that the implementer may freely choose it to help his design. We believe this should change. Design freedoms must be explicitly identified. Silent items that become important during the design phase must be brought back to the attention of the specifiers and resolved early in the development life-cycle. Our enhanced RFC of a specific protocol will do so.

3.5 Improving the Security Robustness of Existing Implementations

A number of formal languages and methods are aimed at helping the protocol designer from the very early stages of formulation of the protocol. In our proposed work, we are interested in taking an existing protocol and an existing implementation and improve the exploit-robustness of the implementation and send feedback in the form of suggested revisions to the corresponding RFC.

3.6 Develop an Approach that is Programmer Oriented

We believe that robustness improvement techniques must be implementer oriented. In the suggestions that result from an improvement techniques, they should be able to immediately recognize holes in their implementation, acknowledge a fix as a solution, and carry out a practical revision of source code.

4 Our Approach

We propose a solution to the above problem (see Section 3.1). We present our approach briefly in this section and in more detail in Section 6. There is just enough detail here to appreciate the relevance and syntactic style of the examples presented in Section 5.

4.1 Overview of Our Approach

First, we fix the specification in such as way that there is no ambiguity for the implementer in those areas where past exploit techniques exist and future exploits can be predicted. Secondly, we classify the "bad" packets in the state-space through formal description techniques. Third, at certain locations in the source code implementation of the protocol, we insert assertions based on this classification. If the platform on which the protocol implementation is running is powerful enough, we can enable the assertions as run-time checks.

A typical RFC defines a protocol using state machines. State transitions are defined only for valid inputs, and on rare occasions deal with a few well-known timeouts. We will introduce additional states defined solely for security purposes.

With the state machine and each state, we attach certain attribute variables. Some of these variables are needed for the normal functioning of the protocol implementation; the remaining variables are present solely to enhance security. The attribute variables can capture timing issues, such as when the packet arrived, real time elapsed in that state, or time spent by the CPU in that state. The attribute variables can capture resource consumption issues, such as number of bytes allocated to various data structures. A declaration attached to a given state divides all the inputs that could be received by that state into, say, G1, G2, ... of "good" inputs, and B1, B2, ... of "bad" inputs. This partitioning is based on the attribute variables, current values of program variables and history of these.

The standard RFC would have defined the behavior of the state machine for the good inputs, but typically would be silent about the bad inputs. In our enhancement of the RFC, we would describe the behavior of the state machine for B1, B2, etc. An implementer would systematically derive source code based on the enhanced RFC (ERFC). Because the ERFC describes the state machine, the attribute variables, and the declarations in a formal language, computer-aided tools that help generate the code from the ERFC become feasible.

4.2 Formal Languages

We describe the attributes and the partitioning of input space not in English but in a language whose syntax and semantics are well-defined, and where software tools can be developed to help verify the claims of improved security.

4.3 Man Power Requirements

The typical implementer of a protocol is an expert on the technical details of the protocol. On the other hand, he may not be security savvy. The approach we are outlining requires a team that has technical expertise on a given protocol, security awareness and training in the use of formal methods. Our approach can take an existing implementation and annotate it with formal methods so that the resulting source code is now exploit-robust.

5 Examples

Before we describe the details of our approach, it is helpful to consider a few short examples. The examples below are written in a C-like language yet to be developed. The examples illustrate how some the common vulnerabilities and attacks can be prevented. We use these later in Section 7 as motivating examples for the ERFC recommendations we make.

We intend to expand considerably the concepts behind this section.

5.1 Wrong Header Values

This example illustates a specific case of packets that do not conform to the RFC (see also Section 7.1).

The range of legal values for the the length of the header is from `len_min` to `len_max`, and the actual value of the same header for a reference packet that we got from the receive queue is `packet.hdr.len`. The current state of the engine is `state_one` and the current state engine will not change to `state_two` unless the header value checks out against legal values for that protocol specification.

The corrective action we illustrate is simpley dropping the offending packet, but it is possible to take more sophisticated actions such as alerting an IDS, or begin gathering more packets that may arrive from the same source address.

```
FDT:
{
  currentState == state_one;
  /* state space */ partitions G, B;
  G = (len_min < packet.hdr.len < len_max);
  B = -G;
}

if (G()) {
  ...existing code...;
  state = state_two;
} else {
FDT:
  drop_packet();
  currentState = state_one;
}
```

5.2 Flooding

This example illustrates *the* essential technique in detecting a flooding attack and related vulnerabilities described in section 7.3.1. That we are being flooded cannot be discovered unless we are aware of the "history of packets" with timestamps.

The `packetHistory` is an abstract value standing for the history of values of the concrete variable named `pkt` that was sampled at the rate of once per 10 ms viewed as a queue of the latest 1000 values.

Reckless use of history will make an implementation infeasible. When used with care, a software tool that we plan to build will help translate history related manipulations into feasible source code.

```
FDT:
{
  currentState == state_one;
  struct Packet packetHistory
      [1000,
       history(pkt, timeDelta == 10 ms),
       queue];
  partitions G, B;
  B = floodHueristic(concreteState,
                     packetHistory);
  G = -B;
};

if (G()) {
  ...existing code...;
} else {
FDT:
  packetHistory = 0;
  currentState = safe_state();
}
```

5.3 Authentication Against Spoofing

The `requiredForAuth` collects the variables required for authentication. The next packet arrival may cause the transition into state-Two. If so, unless the authentication pre-condition of state-Two holds, we should remain in the current state.

```
FDT:
{
  currentState == state_one;
  partitions G, B;

  struct requiredForAuth {
    stateOneTimeout;
    authInformation;
    var2;
  };

  G = (predictNextState() != state_two
       || CheckStateTwoPre(requiredForAuth));
  B = -G;
}

if (G()) {
  ...existing code...;
  state = state_two;
} else {
FDT:
  ...;
}
```

5.4 Header Reuse

This example intends to counter the vulnerabilities related to header reuse (see also Section 7.3.2)

The `packetHistory` is a sequence of identical structs, each having a field named `HdrA`. The `packetHistory.HdrA` then stands for the sequence of just the `HdrA` values. `pktQ` is the queue of packets yet to be processed.

```
FDT:
{
  currentState == state_one;
  struct Packet packetHistory
      [unlimited, history(pkt)];

  HdrAType ha = pktQ.copyAPacket().HdrA;

  partitions G, B;
  B = (ha in packetHistory.HdrA);
  G = -B;
};

if (G()) {
  ...existing_code...;
} else {
```

```
FDT:
  currentState = safe_state();
}
```

5.5 Impact Rollover

This example is about how we can make the specification intelligent about state impact rollover (see Section 7.2 for more details on the corresponding vulnerability). We evaluate what the impact would be on previous states if we were to permit changes of variables in the current state C. For this, we collect all the header values from the packet that we got from the queue (see "statevectors", in Section 6.1) into `MyHdri`. We then consider a roll-back by inserting `MyHdri` values into virtual state dumps of all or any suspicious previous states.

If such roll-back results in a B state, we would drop such packet or packets and force the next state to be a "safe" state.

As can be seen, the idea is simple, but a feasible implementation requires considerable work becuase of active use of history in the form of state dumps.

```
FDT:
{
  currentState == state_one;
  partitions G, B;

  Packet packet = pktQ.copyAPacket();
  Headers myHdri = packet.headers;

  consRolloverImpactState() {
    for i
      in range(0, all_states) {
      substitute(virtual_state_i, myHdri);
      if (virtual_state_i == B);
      return B;
      }
    return G;
  }

};

if (consRolloverImpactState() == G) {
  ...existing code...;
} else {
FDT:
  currentState = safe_state();
  ...;
}
```

5.6 Timeout

This example enforces timeouts for each state into a protocol specification. It checks if the `timeElapsed` since entering state-One against a best practice `stateOneTimeout`.

```
FDT:
{
  currentState == state_one;
  partitions G, B;

  B = timeElapsed(stateOneTimeout);
  G = -B;
};

if (G()) {
  ...existing code...;
} else {
FDT:
  ...;
}
```

5.7 Premature Initiation of States

This example intends to counter the common mistake of "premature initiation of states" (described in section 7.2) by introducing a new state vector called the "state condition list", that needs to be true before the state engine can enter any particular state.

```
FDT:
{
  currentState == state_one;

  struct stateTwoConditionList {
    stateTwoAuthentication;
    valueA;
    valueB;
  };
  partitions G, B;

  B = checkForStateTwoConditions
      (StateTwoConditionList,
       present_state_dump);
  G = -B;
};

if (G()) {
  state = state_two();
} else {
FDT:
  ...;
}
```

6 Details of Our Approach

This section gives some of the details of the proposed solution to the problem of Section 3.1. The first step in our solution is to write an enhancement of the RFC of the given protocol. The next section is devoted to this topic. This section details the formal description techniques (FDT) used. We refer to the examples of the previous section as needed.

Our approach depends heavily on being able to recall the history of values that important variables had. The storage requirements are infeasible if we were to literally record the history. An efficient design for recording history is a research task further discussed in Section 8. History is defined more precisely later in this section.

Our approach also depends on taking a suitably abstract view of the concrete state vector. The abstract view, in particular, removes pointers that are omnipresent in the concrete data structures.

6.1 States and State Vectors

In this section, we will be using the term state in two different ways. The first meaning is that given in state machines. The second usage is as an abbreviation of "state vector" that is a collection of values of all relevant variables of a program at a given control point. There is, of course, a correspondence between these two meanings. Certain control points correspond to the states shown in the state machine. The state vector is the input to the state that will lead a transition into another state.

6.1.1 Snapshots of Concrete State

The concrete state at any moment of an implementation of a protocol is the collection of all variable-value pairs relevant. This includes global variables, contents of certain files, the content of the run-time stack, etc. A quick, if imprecise, description is the concrete state is the "core dump". As the protocol engine executes it makes changes to the concrete state.

A snapshot is a time-stamped record of the concrete state. Snapshots are used in the analyses of an implementation. Snapshots are not necessary, but can be very useful, for the normal functioning of the protocol engine. Note that the size of the snapshot can be very large. Note also that taking snapshots may mean that the protocol engine cannot run in real-time.

6.1.2 Inputs to a State

The input to a state is the concrete state of the engine as it enters the state. It is unlikely that all of the concrete state is relevant to a given state. The output of state is the concrete state resulting as the engine ends the current state and begins the transition to the next state. It is likely that much of the output concrete state is identical to the input concrete state.

6.2 History

History is a sequence of snapshots. The snapshots may be of subsets of concrete state. Obviously, history is immutable.

Real programs do not try to store histories because they can be extremely large. We use it as an abstraction in clearly stating certain vulnerabilities or attacks in progress. Implementation of histories must not be taken for granted.

In the Example 7.3.1 on Flooding, we make use of the history of packets received. For reasons of memory capacity, we limit the history to a total of 1000 items and sampled at a rate of 10 ms.

6.3 Assertions and Abstractions

An assertion is a logical expression on concrete state and histories. In this expression, we may not only use the Boolean and, or, not, ..., but also the $\forall$ "for all" and $\exists$ "there exists" quantifiers. It is possible to convert some (not all) assertions into executable watch-dog code. The Appendix A.1 explains assertions.

The concrete state vector is likely to use various data structures whose details are irrelevant to the issues at hand. For example, that some thing is a linearly linked list is irrelevant. Considering it as a sequence of certain values is more useful. An abstraction function maps the concrete state to a more mathematically amenable abstract model. The abstract models uses sets, sequences, tuples, graphs, etc. but not pointers. The abstraction function is necessary in the analyses. It is a function mathematically well-defined. It is not necessarily programmed into an executable method.

6.4 Improving the Security Robustness

In this section, we describe how one proceeds to improve robustness.

6.4.1 Identifying the States and State Vectors

We need to identify points in the source code that correspond to the states defined in the RFC. We also need to identify the relevant portions of concrete state. This is a task for the implementer. This task is mechanizable using compilation related techniques.

6.4.2 Introduction of Attribute Variables

With the entire state machine and/or with some chosen states, we introduce certain attribute variables. Some of these variables are abstractions of concrete variables needed for the normal functioning of the protocol implementation; the remaining variables are present solely to enhance security. The attribute variables can capture timing issues, such as when the packet arrived, real time elapsed in that state, or time spent by the CPU in that state that are not recorded in concrete variables. The attribute variables can also capture resource consumption issues, such as number of bytes allocated to various data structures.

The snapshots include the attribute variables.

6.4.3 Partitioning the Input Space

The concrete state space is the abstract collection of all possible concrete states. A large subset of these state vectors will not occur simply because of the control flow in the protocol engine. In the paragraph below, we are considering the remaining subset.

We partition the state space into legal and illegal ("should not occur") concrete states from the perspective of the protocol RFC. Note that the illegal states are demonstrably reachable states. Our definition of legality is based on whether the RFC has defined a valid transition for that abstract (mapped from concrete) state.

The good/legal and bad/illegal state spaces are defined using techniques of discrete mathematics and logic. This is a task for the security specialist with the help of a formal methods person.

Frequently, we find a state vector illegal because of the presence of certain values in the history. E.g., the detection of a protocol engine being being flooded (see Section 5.2) depends on recent history.

6.4.4 Introduction of New States

Assuming that the given protocol implementation PI is faithful to the original RFC, we do not expect PI to have transitions corresponding to the illegal concrete states. The code that is present does its job for a good concrete state, but may crash, hang or has other undesirable effects when the bad concrete state is its input.

The ERFC should have introduced new states and defined transitions to them. When an engine is in these states, all further processing should be done with greater care if we are to be exploit-robust. It is helpful for later analyses to begin recording history once a new state is entered.

The ERFC should also have described if, when, and how it is possible to join a good state from one of these bad states.

A new state introduces corresponding new source code. We expect to generate the new source code from the abstract description given in our formal language through the use of a software tool.

6.4.5 Introduction of New Transitions

The transition from a good state X in the existing implementation to a newly introduced state Y occurs when the concrete state is illegal in X. The description of the illegal state is given in the new formal language that we intend to develop. From this description, we intend to generate executable code through interactive use of new software tools that we will develop. The generation of executable code is, in general, a non-trivial problem. We intend to solve it (i) by suitably limiting the expressive power of the new formal language, and (ii) by interactively guiding the code generation software tool.

7 Enhanced RFC

Presently, most of the protocol specifications define what constitute "right" packets relevant to the protocol and they specify what the response should be for such packets. They remain silent on what the protocol implementation should do for packets (meaning header values) which deviate from the specification.

An implementer takes the "silence" in the specification as "design freedom", but usually they have not factored in the security issues.

The first step in our solution is to write an enhancement of the RFC of the given protocol that is not silent on several fronts, and in general, written with an awareness of the "seven sins of a specifier" [Meyer, 1985]. An implementer would systematically derive source code based on the enhanced RFC (ERFC).

The ERFC should present in a formal language the substance of the RFC it intends to subsume. We take this point to be clear, and focus on making a case for why the ERFC should also address the following vulnerabilities caused by insecure implementation of protocol specifications. Specifically the ERFC should consider the following three types of vulnerabilities elaborated in the subsections below.

7.1 Handling of Non-Compliant Protocol Headers

A common exploit technique is to construct packets with headers that are set up to deliberately violate the original RFC.

7.1.1 Protocol field anomalies

This vulnerability arises from insecure handling of illegal protocol header values. A lot of vulnerabilities exist because of handling packets which are non-compliant with protocol specifications, and these types of vulnerabilities are growing lately because of the sophistication of fuzzing techniques and tools.

Example 1: SNMP vulnerabilities [1]

Most of the vulnerabilities arising from the test cases released by OUSPG for SNMP protocol are because the corresponding implementation did not have code

[1] `http://www.cert.org/advisories/CA-2002-03.html`

to handle header values inconsistent with the protocol specification.

Example 2: Malformed H.225.0 messages[2]

The vulnerabilities discovered in the affected products can be easily and repeatedly demonstrated with the use of the OUSPG PROTOS Test Suite for H.323. The largest group of vulnerabilities described in this advisory result from insufficient checking of H.225.0 messages as they are received and processed by an affected system. Malformed H.225.0 messages received by affected systems can cause various parsing and processing functions to fail, which may result in a system crash and reload (or reboot) in most circumstances.

7.1.2 Protocol data anomalies

This vulnerability arises from insecure handling of illegal protocol data following a header.

Most of the time, the determination of whether the protocol data is "good" or "bad" depends on the semantics of the protocol.

Example 1: Vulnerabilities in IKE[3]

"An ISAKMP packet with a malformed payload having a self-reported payload length of zero will cause isakmpd to enter an infinite loop, parsing the same payload over and over again."

"An ISAKMP packet with a malformed IPSEC SA payload will cause isakmpd to read out of bounds and crash."

"An ISAKMP packet with a malformed Cert Request payload will cause an integer underflow, resulting in a failed malloc of a huge amount of memory."

7.1.3 Handling of packet size variations

This vulnerability arises from insecure handling of illegal size of both the protocol header values and the data.

The classic "ping of death" exploit belongs to this category.[4]

7.1.4 Header Number Variations

This vulnerability arises from the protocol specification being silent on the number of inner headers that a legal packet might have.

[2]`http://www.cisco.com/warp/public/707/cisco-sa-20040113-h323.shtml`

[3]`http://www.securityfocus.com/archive/1/358386`

[4]`http://www.insecure.org/sploits/ping-o-death.html`

Example 1: RADIUS[5]

According to RFC 2865, each RADIUS packet can be up to 4096 bytes. It allows packing > 2000 attributes into a single packet. Most RADIUS servers' implementations allocate maximum attribute length for each attributes, it means for each attributes > 256 bytes of memory will be allocated. It is possible to lock about 512K of memory and waste CPU time with a single 4K packet resulting in an easy denial of service attack.

Example 2: Malformed ISAKMP Delete[6]

"An ISAKMP packet with a malformed delete payload having a large number of SPIs will cause isakmpd to read out of bounds and crash."

7.1.5 Insecure interpretation of special and reserved packets

This vulnerability arises from insecure handling of special and reserved packets, or the protocol specification being silent on some special flags in the protocol.

All the vulnerabilities related to handling of IP options, fragmentation, etc. fall into this category. When there are header fields, turning them off or on can change the processing of a packet carrying those fields. The RFC needs to carefully specify what do do when the options are absent. This exploit also refers to protocol specifications being silent on how to handle special scenarios such as loop back packets, multicast packets etc.

Examples include "teardrop" (overlapping IP fragmentation re-assembly bug)[7] the "land" attack (source address and port are spoofed to be the same as the destination)[8], and devious IP options[9].

7.2 Insecure Protocol State Handling

This vulnerability refers to inconsistent and insecure state handling in the protocol stack. Here are some examples that fall into this category.

Premature initiation of states: This vulnerability refers to entering a state (sometimes initiating a process corresponding to that new state) before it is so needed.

Unauthenticated (or "illegal") state entry: This vulnerability refers to entering into a state based on unauthenticated packets from untrusted source.

[5]`http://www.securiteam.com/exploits/6K00J2035U.html`

[6]CVE ID: CAN-2004-0221

[7]`CERT Advisory CA-1997-28`

[8]`CERT Advisory CA-1997-28, also http://www.cisco.com/warp/public/770/land-pub.shtml`

[9]`http://www.securiteam.com/unixfocus/IP_options_processing_Denial_of_Service_in_BSD.html`

State impact roll-over: This vulnerability refers to the impact on a state rolling over and impacting the previous states as well.

A well-known example of this type of exploit is the SYN floods on a port affecting the already ESTABLISHED connections on a port.

Insecure state exit: This vulnerability refers to not reclaiming the resources (memory etc.) before exiting a state.

Numerous "memory leak exploits" fall into this category[10].

7.3 Handling Attack Techniques

We also expect the RFC to handle common attack techniques, including the following:

7.3.1 Flooding of protocol packets

This vulnerability arises when the specification and implementation does not handle large volume of packets in a secure way. Since this is a common attack technique, the ERFC should address it.

7.3.2 Packet replay and Header Reuse

This vulnerability refers to the protocol implementation not anticipating and recognizing the reuse of protocol headers and replaying of packets belonging to an earlier communication.

Protocols that have session-based authentication are vulnerable to packet replay attacks. E.g., SSH[11], Kerberos[12], and TACACS [13] have all been exploited with replay.

8 Research Tasks

We intend to develop a new language for use in both ERFC and in the FDT code insertions into existing source code. The design specification language defined in [Mateti, 1990], the network protocol implementation language Prolac [Kohler *et al.*, 1999], and [ASN, 2002] are our starting points for the definition of our formal language.

8.1 Functional Requirements

The language should have ways to represent

- protocol headers (types, sizes, etc.)
- states (description, attributes, etc.)
- state transitions

Luckily, almost all of the existing languages, such as SDL and Prolac, have these features. But, nearly all of these languages fail to satisfy the "human factors" requirements described below.

8.2 Human Factors Requirements

- There should be no ambiguity in what is being described.
- The language should be easy for an implementer to understand and implement.

And, one can list several more requirements. But, the above are sufficient to make the point that requirements in this category are going to be nebulous but very important. If we were to ask ourselves, why the impact of formal methods and languages has been this low in spite of active work over the past two decades, the only answer that we can give is that the language was too "foreign" to the implementers. Note that ambiguity can arise only after comprehension.

Of the languages we looked at, XML/XSLT/Java based language of [Abdullah and Menasc, 2003], and C/Lisp/Scheme based language called Prolac of [Kohler *et al.*, 1999] are both acceptable in this regard.

8.3 Manipulations on History

In order to prevent various exploits, such as flooding, from occurring, any implementation has to record portions of history and re-examine upon suspicion of flooding. Consequently, our approach depends heavily on being able to recall the history of values that important variables had. On the other hand, the storage requirements are infeasible if we were to literally record the history.

In our language manipulations on history are expressed in a functional-programming style at a very high level.

8.4 Code Generation

We expect to be able to generate C-like code, from the functional programming constructs used in the FDT, that is realistic in terms of resources through the interactive use of a software tool to be developed.

There is a sizable body of literature (e.g., [Castelluccia *et al.*, 1997], [Abdullah and Menasc, 2003] [McGuire, 2004a], [McGuire, 2004b]) on this topic.

[10] http://www.cisco.com/warp/public/707/catalyst-memleak-pub.shtml

[11] http://www.kb.cert.org/vuls/id/565052

[12] http://www.hut.fi/ãutikkan/kerberos/docs/phase1/pdf/LATEST_replay_attack.pdf

[13] http://www-arc.com/sara/cve/tacacs_server.html

8.5 Secure Programming Know-How

Formal methods will continue to look too foreign unless we employ the only available solution, namely "engineering education". Implementers need to be educated on good protocol implementation procedures and "strict checking and parsing" of incoming packets.

We propose to write a tutorial in the context of IKE.

9 Deliverables

9.1 Enhanced RFC for A Protocol

We plan to take a protocol, and re-write the RFC in our secure way into an ERFC. Our current choice of protocol for this experiment is IKE.

9.2 Vulnerabilities Discovered in the Protocol

The IKE protocol has been the subject of many analyses (see, e.g., [Zhou, 2000], [Hallqvist and Keromytis, 2000], and [Song *et al.*, 2001]). We will objectively re-examine if any of these results would have been discovered in the process of developing the ERFC. We will present all vulnerabilities found in the protocol based on standard RFC.

9.3 Vulnerabilities Discovered in an Implementation of the Protocol

We wish to "prove" in some way that had we given the implementers the ERFC and the FDT enhanced source code, some of the implementation vulnerabilities would have been eliminated

9.4 IETF RFC

We will submit the ERFC template of ours to IETF suggesting its use as a clarification instrument if not the defining document.

9.5 Draft Report Defining Our Formal Language

We will write a report defining our formal language. The report will have grammar and prose descriptions, many examples of its usage and other commentary.

9.6 Suggestions about How the Methodology Applies to Other Protocols

We expect our methodology to be applicable to protocols other than IKE. We will maintain a blog of our proposed work, and will include suggestions about how our methodology is more universally applicable.

10 Related Work

In this section, we summarize related work that did not get mentioned in other sections.

10.1 Formal Methods and Languages For Protocol Analysis

Formal languages have long been used for assessing the security of computer systems [Wing, 1998]. There is a large body of literature that applies formal methods to the verifying the correctness and efficiency of protocols [Berry and Gonthier, 1992] [Basu *et al.*, 1997], and formal analysis and design verification of crypto protocols [Saul and Hutchison, 1999], [Broy, 1991].

[Wing, 1990] is an introduction to specifying with formal methods. [Older and Chin, 2002] is a good introduction to how formal methods are applied to the analyses of secure protocols. [Babich and Deotto, 2002] is an introduction to formal languages developed for protocol descriptions. This article gives a birds eye view of SDL, Promela and Lotos.

Interestingly enough, we could not find any past work, related to removing the ambiguity of protocol specifications through formal methods or improving the robustness of specifications against common attack techniques.

10.2 Attack Techniques Against Protocol Implementations

There is considerable progress in the last few years in testing of protocol implementations. Automated tools based on testing techniques known as "fuzzing", "fault injection" and "boundary testing" are becoming more common. [Dawson *et al.*, 1997] [PROTOS, 2004] [Vasan and Memon, 2004]. There are sophisticated techniques for generating test cases as well [Sinha and Suri, 1999] [Kaksonen *et al.*, 2001].

There are even a few attack tools available on this technique. Spike API, a fuzzer creation kit[14], is one such tool concentrating on HTTP protocol. There are commercial sophisticated fuzzing tools like Codenomicon [15] and the tools which are privately developed, like the one used in BGP Vulnerability Testing[16] and we believe that sophisticated open source protocol fuzzing tools are around the corner.

10.3 Code Analysis Tools

SPLINT [Evans, 2002] is a compile-time tool that analyzes C source code based on formal annotations given

[14]`http://www.immunitysec.com/ resources-freesoftware.shtml`

[15]`http://www.codenomicon.com/`

[16]`http://www.nanog.org/mtg-0306/pdf/franz.pdf`

by the programmer, and can pin point many errors. This is a further development of an earlier tool called Larch. [Larochelle and Evans, 2001] and [Evans and Larochelle, 2002] are examples of how the SPLINT tool helped locate buffer-overflow problems in `wu-ftpd`.

Using a technique called meta-compilation, Engler and his group have been successful in identifying bugs in software systems ([Musuvathi and Engler, 2004], [Chou *et al.*, 2001], [Chelf *et al.*, 2002], [Hallem *et al.*, 2002]).

The NRL protocol analyzer has been used by many including including its developer ([Meadows, 1996],[Meadows, 1999],[Meadows, 2003]). There is considerable activity on CAPSL [Millen, 2004].

11 Discussion

In this section, we present answers to some questions we imagined or heard in the context this proposal.

11.1 Are Formal Languages the Best Solution?

From our perspective, the answer is "Yes." Recall that C, C++, Intel Assembler language are all formal languages. Using suitable formal languages in protocol design and implementation makes it possible to "compile" a high-level description into feasible code. Today we do not debate whether a skilled assembly programmer can produce better code for an algorithm described in C. We only debate the productivity of such an endeavor.

11.2 What "Protocol Problems" Did Formal Languages Solve?

Let us defuse the question by answering "None." We are not being Clintonian, but it really depends on what is meant by "solve."

E.g., did formal languages/methods help find flaws in protocols after the protocol is some how designed, formulated and implemented? Answer: yes. (Right now I am short of time to search for exact citations, but SDL, Lotos, etc. were used in this kind of work.) Did formal methods help find bugs and vulnerabilities? Yes. Did formal methods help generate "acceptably good" implementations of protocols from formal descriptions of the protocol? Yes. Did formal methods help security-harden existing implementations of protocols? We do not know of such work. Can formal methods help generate differing implementations intended to run on platforms with varying degrees of computing power? Yes. Can we address device specific issues like CPU inside the specification? Yes. Some protocols(especially public protocols like web, mail) require accepting a packet from any client. In that case, in order to implement the "Flood Heuristic" function above, we need to take the device's CPU level and memory consumption into consideration. It is another question whether our ERFC should talk about what constitutes such heuristics. We would argue that we should, because we are trying to solve the problem of "non security-savvy" implementer.

11.3 Must We Design Our Own Formal Language?

Many formal languages are gratuitously difficult. Using a notation and semantics already familiar to implementers has not been a concern of the designers of languages like Lotos. This is not to say that by merely making the notation C-like all problems of comprehension and training are solved.

11.4 Can Formal Methods Replace Testing?

Testing and formal methods should be seen as being orthogonal. To repeat a famous quote, "Testing reveals the presence of bugs not their absence." Formal methods typically focus on guaranteeing the absence of bugs. However, in this proposal our focus was in taking a properly specified RFC and a legacy implementation, and generate – via interactive use of formal methods tools – executable code.

11.5 Are We Doing Intrusion Detection and/or Prevention?

Checking that packets are indeed obeying the rules of the protocol and only then processing them, otherwise taking some corrective actions smacks of intrusion prevention. Our point of view here is that it is like suggesting that an operating system should check that `sqrt()` library function is not invoked with a negative argument. A protocol engine must take responsibility for its inputs. The engine must remain robust for all inputs that it cannot guarantee will not happen.

11.6 Will RFC Writers Use ERFC to Write All RFCs in The Immediate Future?

Almost nil. We consider the project a success even if we were only able to get IETF accept ERFC as an "informational RFC".

Whether or not RFC writers will be persuaded to use ERFC like languages will depend on the need to do so. We guess that sophisticated open source fuzzing tools would expand such a need.

Since there are no other solutions now, we want to put our solution out there. We hope that it will be refined and developed as per the community needs and will be ready when such a time comes.

11.7 Should The RFC Writers Be Aware of Attack Techniques?

Yes. One of the goals of the project is to make the specification aware of the attack techniques so that the implementers do not have to worry about them.

Furthermore, the "attack techniques" do not change from protocol to protocol and it would not be hard to acquire such knowledge through the example ERFC that we would be writing. Note that even though the attack techniques don't change, the replies (drop, reject, error, or safe State) would change as per the requirements of various protocols.

12 Conclusion

A large number of protocol implementations can be impaired by sending them cleverly crafted sequence of packets. Such vulnerabilities occur because the corresponding implementation failed to anticipate packets that do not correspond to the protocol specification.

In this document, we proposed a solution to this problem consisting of the following steps. (i) We fix the specification as given in the original RFC in such a way that there is no ambiguity for the implementer in those areas where past exploit techniques exist and future exploits can be predicted. We call the resulting document an Enhanced RFC. (ii) We classify the "bad" packets in the state-space through formal description techniques. (iii) At certain locations in the source code implementation of the protocol, we insert assertions based on this classification. If the platform on which the protocol implementation is running is powerful enough, we can enable the assertions as run-time checks.

Our approach depends on a careful use of the history of the concrete state vector of the protocol engine, and progressively refining into a feasible implementation. Our approach makes use of advances in the design of very-high level programming languages, compilation techniques, formal methods, and security improvement techniques.

A Tutorial on Formal Methods

[Dijkstra, 1976], [Gries, 1981], and [Björner and Jones, 1982] are three classic books that can together give an excellent background to a practical programmer. Below we give a whirlwind summary of definitions and examples of some of the terms used in this proposal.

A.1 Assertions

An assertion is a logical expression on the variables of a program. E.g.,

$$x - \delta_1 \leq (sqrt(x))^2 \leq x + \delta_2,$$

for some small values δ_1 and δ_2, is an assertion.

The logical expression may use well-known mathematical and programmer-defined functions. The expression must not have side-effects. In a C program that has a struct variable named `ws`, a relevant assertion is:

```
(ws == NULL ||
 (ws->nc <= ws->sz && ws->ps != NULL
  && ws->ns + 1 == nnulc(ws->ps, 1 + ws->nc)))
```

where `nnulc()` is a function defined elsewhere by the programmer.

The expression can also use quantifiers. E.g., at the end of an algorithm sorting an array $a[1:n]$ of numbers, we expect to have

$$\forall i : 1 \leq i < n \rightarrow a[i] \leq a[i+1] \wedge bag(a) = bag(a.old).$$

A.2 Pre- and Post-conditions

A pre-condition is an assertion that is expected to be true immediately upon entering a procedure/ function/ method. The bod of the method assumes this and proceeds to accomplish its task. A post-condition is an assertion that describes what the procedure will accomplish. A pre-condition involves parameters to the procedure and globals. It must never involve variables local to the procedure. A post-condition involves the current values of the parameters, globals, and local variables, and relates them to the values the globals had upon entry.

A.3 Invariants

An invariant is an assertion whose value remains true every time control reaches it. A loop invariant is inserted at a location inside a loop. A class invariant must be true as both pre- and post-condition of every public method of a class.

A.4 Abstraction Function

An abstraction function drops irrelevant detail from concrete data structures of a program. E.g., a program may be implementing a set using an array `int a[100]`, whose abstraction function maps the array into the set it represents.

```
abstract(a) = { a[i] : 0 <= i < 100 };
```

B Bertrand Meyer's Seven Sins of the Specifier

B.1 Noise

The presence in the text of an element that does not carry information relevant to any feature of the problem. Variants: redundancy; remorse.

B.2 Silence

The existence of a feature of the problem that is not covered by any element of the text.

B.3 Overspecification

The presence in the text of an element that corresponds not to a feature of the problem but to features of a possible solution.

B.4 Contradiction

The presence in the text of two or more elements that define a feature of the system in an incompatible way.

B.5 Ambiguity

The presence in the text of an element that makes it possible to interpret a feature of the problem in at least two different ways.

B.6 Forward reference

The presence in the text of an element that uses features of the problem not defined until later in the text.

B.7 Wishful thinking

The presence in the text of an element that defines a feature of the problem in such a way that a candidate solution cannot realistically be validated with respect to this feature.

References

[Abdullah and Menasc, 2003] Ibrahim S. Abdullah and Daniel A. Menasc. Protocol specification and automatic implementation using XML and CBSE. In *Proc. of Int. Conf. on Communications, Internet and Information Tech. (CIIT2003)*, page 6pp, Scottsdale, AZ, 2003.

[ASN, 2002] ASN. Abstract Syntax Notation One (ASN.1) standards, 2002. `asn1.elibel.tm.fr/en/standards/`.

[Babich and Deotto, 2002] Fulvio Babich and Lia Deotto. Formal methods for specification and analysis of communication protocols. *IEEE Communications Surveys and Tutorials*, 4(1):00–00, 2002. `www.comsoc.org/livepubs/surveys/public/2002/dec/ index.html`.

[Basu *et al.*, 1997] Anindya Basu, Greg Morrisett, Mark Hayden, and Thorsten Eicken. A language-based approach to protocol construction. In *Proc. ACM SIGPLAN Workshop on Domain Specific Languages*, January 1997.

[Berry and Gonthier, 1992] Gerard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[Björner and Jones, 1982] Dines Björner and Cliff B. Jones. *Formal Specification & Software Development.* Prentice-Hall, 1982. ISBN 0-13-329003-4.

[Broy, 1991] Manfred Broy. Towards a formal foundation of the specification and description language SDL. *Formal Aspects of Computing*, 3(1):21–57, 1991.

[Castelluccia *et al.*, 1997] Claude Castelluccia, Walid Dabbous, and Sean O'Malley. Generating efficient protocol code from an abstract specification. *IEEE/ACM Trans. Networking*, 5(4):514–524, 1997.

[Chelf *et al.*, 2002] Benjamin Chelf, Seth Hallem, and Dawson Engler. How to write system-specific, static checkers in metal. In *PASTE 2002*, 2002.

[Chou *et al.*, 2001] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *SOSP 01*, 2001.

[Dawson *et al.*, 1997] Scott Dawson, Farnam Jahanian, and Todd Mitton. Experiments on six commercial TCP implementations using a software fault injection tool. *Software: Practice and Experience*, 1(1):1–26, January 01 1997.

[Dijkstra, 1976] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[Evans and Larochelle, 2002] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, Jan/Feb 2002. `http://www.cs.virginia.edu/~evans/pubs/ieeesoftware.pdf`.

[Evans, 2002] David Evans. Splint - Secure Programming Lint. Technical report, University of Virginia, 2002. `http://www.splint. org`.

[Gries, 1981] David Gries. *The Science of Programming.* Springer, New York, 1981.

[Hallem *et al.*, 2002] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *PLDI 2002*, 2002.

[Hallqvist and Keromytis, 2000] Niklas Hallqvist and Angelos D. Keromytis. Implementing Internet Key Exchange (IKE). In *Proceedings of the USENIX 2000 Annual Technical Conference, Freenix Track*, pages 201 – 214, San Diego, CA., June 2000.

[Kaksonen *et al.*, 2001] R. Kaksonen, M. Laakso, and A. Takanen. Software security assessment through specification mutations and fault injection. In *Communications and Multimedia Security Issues of the New Century / IFIP TC6/TC11 Fifth Joint Working Conference on Communications and Multimedia Security (CMS'01)*, 2001. `http://www.ee.oulu.fi/research/ouspg/protos/analysis/CMS2001-spec-centered/presentation.pdf`.

[Kohler *et al.*, 1999] Eddie Kohler, M. Frans Kaashoek, and David R. Montgomery. A readable TCP in the Prolac protocol language. In *ACM SIGCOMM '99 Conference: Applications, Technologies, Architectures, and Protocols for Computer Communication,*, pages 3–13, Cambridge, Massachusetts, USA, August 1999.

[Larochelle and Evans, 2001] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the Tenth USENIX Security Symposium*, pages 00–13, Washington, D. C., August 2001. USENIX. `http://www.usenix.org/publications/library/proceedings/sec01/full_papers/larochelle/larochelle.pdf`.

[Mateti, 1990] Prabhaker Mateti. ôm: a design specification language. Technical Report WSU-CS-90-07, Wright State U, Jan 1990.

[McGuire, 2004a] Tommy M. McGuire. *The Austin Protocol Compiler Reference Manual*, May 2004. `www.cs.utexas.edu/users/mcguire/software/apc/reference/reference.html`.

[McGuire, 2004b] Tommy M. McGuire. *Correct Implementation of Network Protocols.* PhD thesis, The University of Texas at Austin, 2004.

[Meadows, 1996] C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming,*, 26(2):113–131, 1996.

[Meadows, 1999] C. Meadows. Analysis of the Internet Key Exchange protocol using the NRL protocol analyzer. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, May 1999.

[Meadows, 2003] Catherine Meadows. Formal methods for cryptographic protocol analysis: Emerging issues and trends. *IEEE Journal on Selected Areas in Communication*, 21(1):44–54, Jan 2003.

[Meyer, 1985] Bertrand Meyer. On formalism in specifications. *IEEE Software*, 2(1):6–26, January 1985.

[Millen, 2004] J. Millen. CAPSL Common Authentication Protocol Specification Language Web site. `http://www.csl.sri.com/users/millen/capsl`, 2004.

[Musuvathi and Engler, 2004] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *USENIX Symposium on Networked Systems Design and Implementation NSDI 2004*, San Francisco, CA 94108, 2004. USENIX.

[Older and Chin, 2002] Susan Older and Shiu-Kai Chin. Formal methods for assuring security of protocols. *The Computer Journal*, 45(1):46–54, ???? 2002.

[PROTOS, 2004] PROTOS. Security testing of protocol implementations, 2004. `http:// www.ee.oulu.fi/ research/ ouspg/ protos/ index.html`.

[Saul and Hutchison, 1999] E. Saul and A.C.M. Hutchison. SPEAR II: The security protocol engineering and analysis resource. In *Second Annual South African Telecommunications, Networks and Applications Conference*, pages 171–177, Durban, South Africa, September 1999.

[Sinha and Suri, 1999] Purnendu Sinha and Neeraj Suri. Identification of test cases using a formal approach. In *Symposium on Fault-Tolerant Computing*, pages 314–321, 1999.

[Song *et al.*, 2001] Dawn Song, Sergey Berezin, and Adrian Perrig. Athena, a novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1,2):47–74, 2001.

[Vasan and Memon, 2004] Arunchandar Vasan and Atif M. Memon. Aspire: Automated systematic protocol implementation robustness evaluation. In *Australian Software Engineering Conference (ASWEC 2004), Melbourne, Australia*, 2004.

[Wing, 1990] Jeanette M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8, 10–22, 24, sep 1990.

[Wing, 1998] Jeannette M. Wing. A symbiotic relationship between formal methods and security. In *Proceedings from Workshops on Computer Security, Fault Tolerance, and Software Assurance: From Needs to Solution, CMU-CS-98-188*, December 1998.

[Zhou, 2000] J. Zhou. Further analysis of the Internet Key Exchange protocol. *Computer Communications*, 23:1606–1612, 2000.

C Biodata of Prabhaker Mateti

Ph. D., 1976, Computer Science, University of Illinois at Urbana-Champaign.
`http://www.cs.wright.edu/~pmateti/`

I teach courses on Internet Security, Distributed Computing, Operating Systems, and Software Engineering. My research interests are Internet Security, Formal Methods, Distributed Computing, Software Engineering, and Language Design. I designed a specification and design language called OM. I have directed several MS theses, and three PhD dissertations.

Employment

- Associate Professor, Department of Computer Science & Engineering, Wright State University, 1988 – present.
- Associate Professor, Department of Computer Engineering & Science, Case Western Reserve University, 1981 – 1988.
- Visiting Associate Professor, Department of Computer Science, Washington State University, 1980 – 1981.
- Lecturer, Department of Computer Science, University of Melbourne, Australia, 1976 – 1981.
- Instructor, Computer Science, University of Texas at Austin, 1975 – 1976.

Selected Publications

- Prabhaker Mateti, "TCP/IP Suite," in *The Internet Encyclopedia*, Bidgoli (Ed), John Wiley & Sons, ISBN: 0471222011, December 2003.
- Prabhaker Mateti, "Hacking Techniques in Wireless Networks," invited contribution to *The Handbook of Information Security*, John Wiley, to appear.
- Prabhaker Mateti, "A Laboratory Course on Internet Security," Proceedings of the 34th ACM SIGCSE technical symposium on Computer Science Education, Reno, Nevada, USA, 2003, pages 252 - 256, ISBN:1-58113-648-X, ACM Press New York, NY, USA.
- Chen Ding and Prabhaker Mateti, "A Framework for the Automated Drawing of Data Structure Diagrams," IEEE Transactions on Software Engineering, Vol. 16, No. 5, May 1990, 543-557.
- Prabhaker Mateti, and Ravi Manghirmalani, "Morris' Tree Traversal Algorithm Reconsidered,", Science of Computer Programming, Vol. 11, 29-43, 1988.
- Prabhaker Mateti, "Pascal Versus C : A Subjective Comparison," Springer-Verlag's Lecture Notes in Computer Science, Vol. 79, pp. 37-69. (Also reprinted in Comparing and Assessing Programming Languages: Ada, C and Pascal, Prentice-Hall 1984.)
- Prabhaker Mateti, "A Specification Schema for Indenting Programs," Software - Practice and Experience, Vol.13, 163-179, 1983. (Reprinted in Software Specification Techniques, McGettrick and Gehani (Eds.), Addison-Wesley 1985.)
- Prabhaker Mateti, "A Decision Procedure for the Correctness of a Class of Programs," Journal of ACM, Vol. 28, No.2, pp. 215-232, April 1981.

Synergistic Activities

- A pilot course aimed at undergraduate seniors CEG 499: Internet Security was developed and taught for the first time. The course has now become a regular course numbered CEG 429 and is offered twice an year, typically attracting about 50 students, and is an elective course in the two degree programs BS in CS and BS in Computer Engineering. Detailed lectures of this course are on the web:

  `www.cs.wright.edu/~pmateti/InternetSecurity`
- Panelist, NSF Graduate Research Fellowships Program, 2000-2002.
- Panelist, NSF DUE CCLI, 1998-2000.
- Supervised minority graduate students, including several females.

- Department lead in preparing for ABET EAC and CAC visits.
- Chair, Undergraduate Studies Committee.

Graduate Students

Ph D. Students (3): Amitava Datta (1992; Siemens Research), Chen Ding (1990; Independent), Kouakau Diby (1990, Ciba-Giegy).
MS Students (22): Sunil S. Gadi, Sripriya Subramanian, Venkat Pothamsetty, Laura Daniels, Richard L. Hollenbach II, C. N. Ravikiran, David G. Ferguson, Yain Miou Chen, Himanshu Rawell, Joy Kung, Bo Liu, Kan Zhao, Ilies Bougambouz, Kouakau Diby, Joseph Warzecha, Mohamed Boumaza, Ravi Manghirmalani, Fransisco Ojeda, Yoshiharu Kato, Charles Heller, Jay Patel, Joxan Jaffar.

D Budget

The numbers below are unofficial, but are expected to be fairly accurate. Our Office of Sponsored Research and Development will compute the official numbers.

Period of research: Sept 2004 - Aug 2005.

Mateti's salary (3 months, 14.8% fringe included)	\$ 30,500
One MS/PhD Student (4 quarters)	\$ 16,500
One BS student (4 quarters)	\$ 14,500
Travel	\$ 5,000
Non-Cisco Equipment	\$ 5,000
Total	\$ 71,500

All the above include University overhead dollars. Mateti will work on this research exclusively during the three summer months. The students will work as research assistants.

July 22, 2004