Automatically Patching Errors in Deployed Software

Authors - Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin Michael D. Ernst, and Martin Rinard

presenter name(s) removed for FERPA considerations

Problem Definition

Possible Attacks

Buffer Overruns, Illegal Control Transfers, Other potential Incorrect Behavior

Standard Mitigation

Terminate the application (loss of data, error persists, restart overhead)

Alternative

Automatically Patch the application

ClearView

- Protect against unknown vulnerabilities
- Preserve functionality
- Do not modify source code
- Do not require any cooperation from developers

Introduction

Usual behavior after software failure: DoS

What can we do if we want the application to be available in spite of failures?

Hint: invariants

Suggested solution: Force invariants.

Summarizing Clearview's Automatic patching mechanism

- Learning
- Monitoring
- Correlated Invariant Identification
- Candidate Repair Generation
- Candidate Repair Evaluation

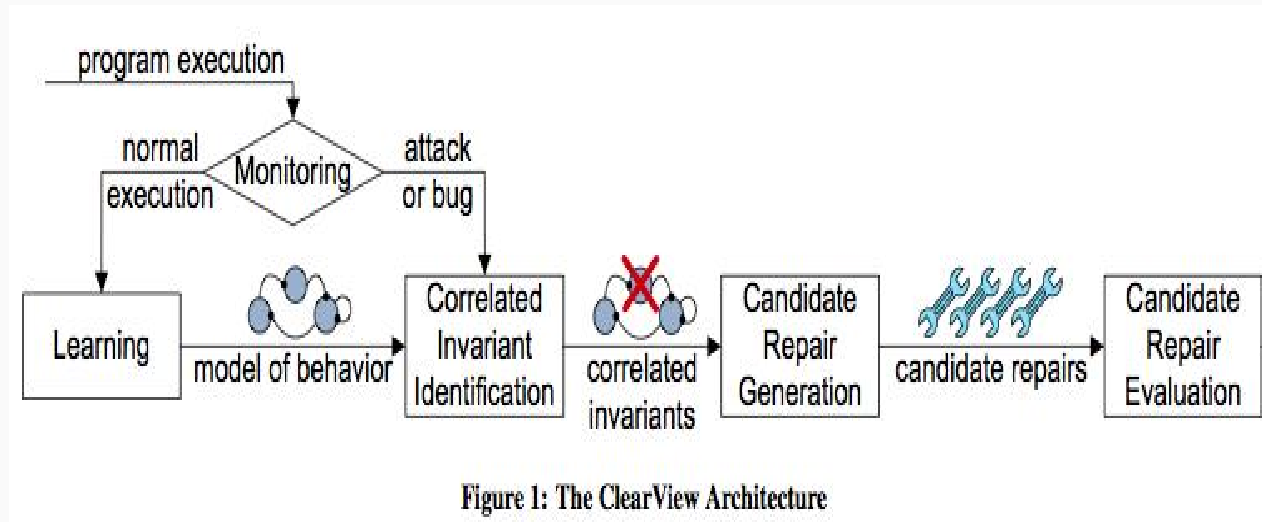


Figure 1: The ClearView Architecture

An example of clearview working

```
@@ -1554,6 +1554,10 @@
     if (wordLen > n) {
       wordLen = n;
     }
+     // WordLen should never be negative.
+     if (wordLen < 0)
+       wordLen = 0;
+
     inWord = PR_FALSE;
     if (isWhitespace) {
       if ('\t' == bp[0]) {
```

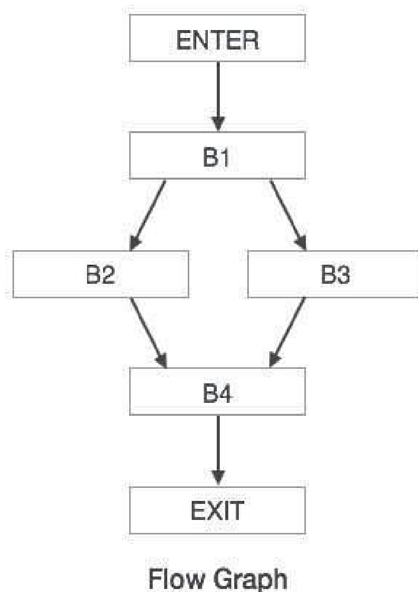
Patch generated

Learning Invariants

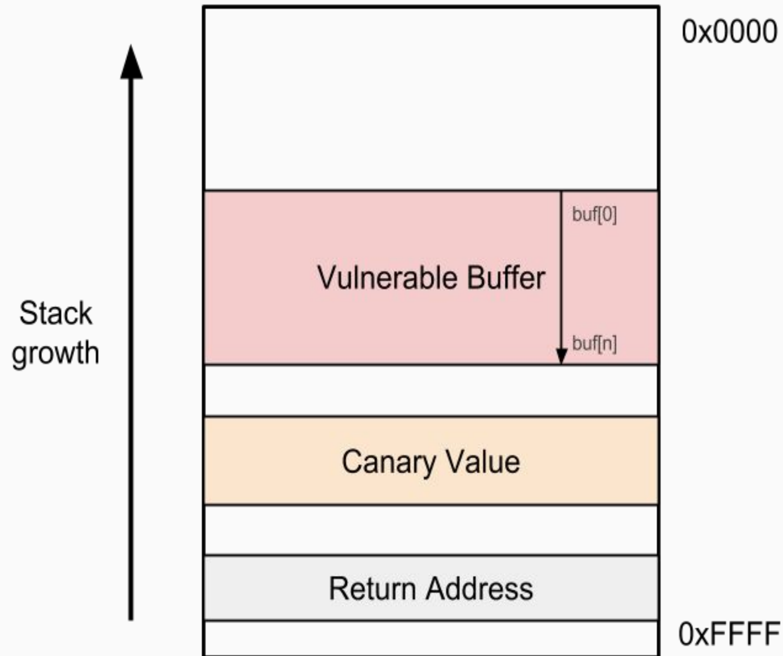
- Create a model consisting of invariants observed during normal execution.
- How to obtain invariants? USE: Daikon

Procedure Control Graphs

- Create a control flow graph. (Done by Determina Program Execution Environment)
- Determine values of variables within basic blocks.
- Use those values to get invariants.



Monitoring



Current Implementation uses MemoryFirewall to detect illegal ControlFlowTransfer errors and it is always enabled.

HeapGuard monitor detects OutOfBounds memory accesses by placing canary values at the boundaries of allocated memory blocks.

HeapGuard

- HeapGuard encounters a canary value
  - Searches an allocation map
    - if address within bounds, normal execution
    - else Out of Bounds write error.
- HeapGuard suffers no false positives
- HeapGuard can detect an earlier error than MemoryFirewall and hence enhances ClearView's ability to find a successful patch earlier.
- HeapGuard can be turned on and off dynamically while the application executes.

Example

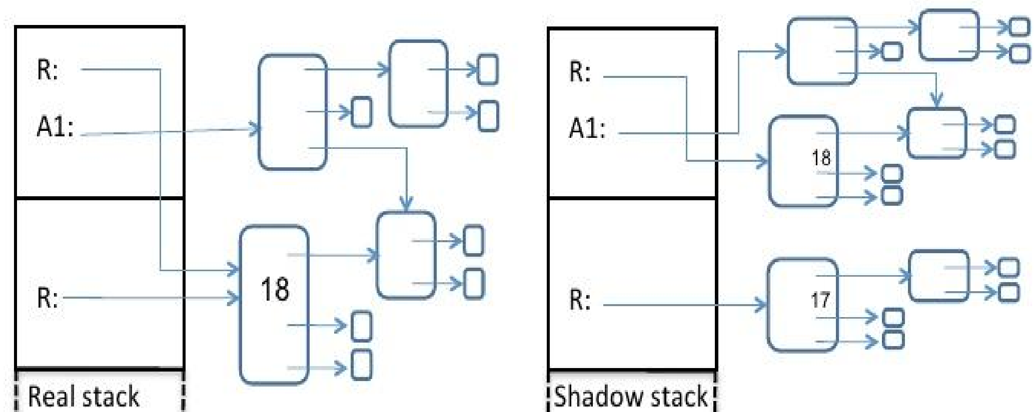
```
int func(){
   int pass = 0;
   char buffer[15];

   printf("\n Enter the password : \n");
       gets(buffer);
Canary value

   if(strcmp(buffer, "password")){
       printf ("\n Wrong Password \n");
}
else{
   printf ("\n Correct Password \n");
   pass = 1;
}
if(pass){
   printf ("\n Root privileges granted \n");
 }
}
```

Monitoring - Shadow Stack

Why?

- Various optimizations can make it difficult to reliably traverse a native call stack.
- Errors such as buffer overflows may corrupt the native call stack and it maybe unavailable to ClearView on a failure detection.

This can also be enabled or disabled when the application is running.



Correlated Invariant Identification

Properties

- Always satisfied in normal executions but violated in erroneous executions
- Violated before failure occurs

Rationale

- May correct the error
- Eliminate the failure
- Enable the application to operate successfully

Candidate Correlated Invariants

Use Shadow stack

Or

Use instructions close to failure location.

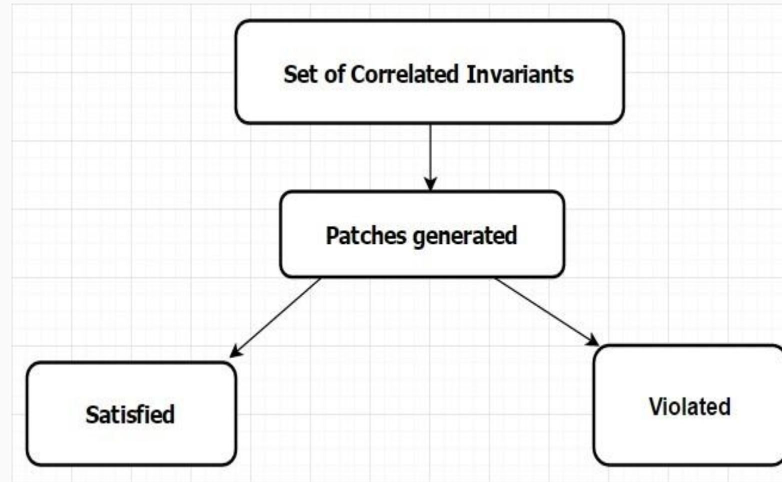
Three key considerations

1. Location of the failure
2. Sufficiently large invariant search space
3. Keeping the set tractable

How to limit the invariant set?

Values of two variables to be compared should be in the instruction’s basic block.

Checking Candidate Correlated Invariants



Single Variable Patches - Execute when the program counter reaches the instruction associated with the variable.

Two Variable Patches - Execute when the program counter reaches the second instruction.

Identifying Correlated Invariants

When a monitor detects a failure, ClearView uses the sequences of invariant checking observations to classify candidate correlated invariants as follows:

- Highly Correlated
- Moderately Correlated
- Slightly Correlated
- Not Correlated

Candidate Repair Generation

Creates a set of candidates repair for each correlated invariant. After checking to see if the invariant is violated, clearview patch will enforce the invariant by changing:

- Flow of control
- Values of registers
- Values of memory location

Candidate Repair Evaluation

After applying the patches, many of them might have negative or no effect on the application.

To solve this issue, clearView observes the application as it executes and ranks each patches based on whether them it observes any failure or crashes

At each point it applies the highest ranked patch in an attempt to minimize the likelihood of a negative effect

Red Team Exercise

- DARPA hired ten engineers from Sparta Inc. to perform Red team exercise.
- 10 exploits in Firefox 1.0.0 (x86 binary) were used to design attacks.

Preparation for Red team exercise:

- Red team has access to all materials generated by Blue team (they generated source code, documentation, analyses of vulnerabilities.)
- Invariant database created by limiting learning to just areas of applications related to vulnerabilities.

Results of Red team exercise

- Clearview succeeded in detecting all exploits and preventing them.
- But, Clearview produced successful patches only in 7 cases out of the 10 attacks.
- Clearview was not affected by false positives.
- Clearview successfully handled variants of attacks and mixed attacks.

3 Cases of failure:

- Misconfiguration of ClearView.
- Invariant obtained was not statistically significant.
- Daikon could not detect appropriate invariant.

Performance

<u>Learning Overhead: -</u>

Time required to load 12 learning webpages

Without learning enabled - 5.2 seconds

With learning enabled - 1600 seconds

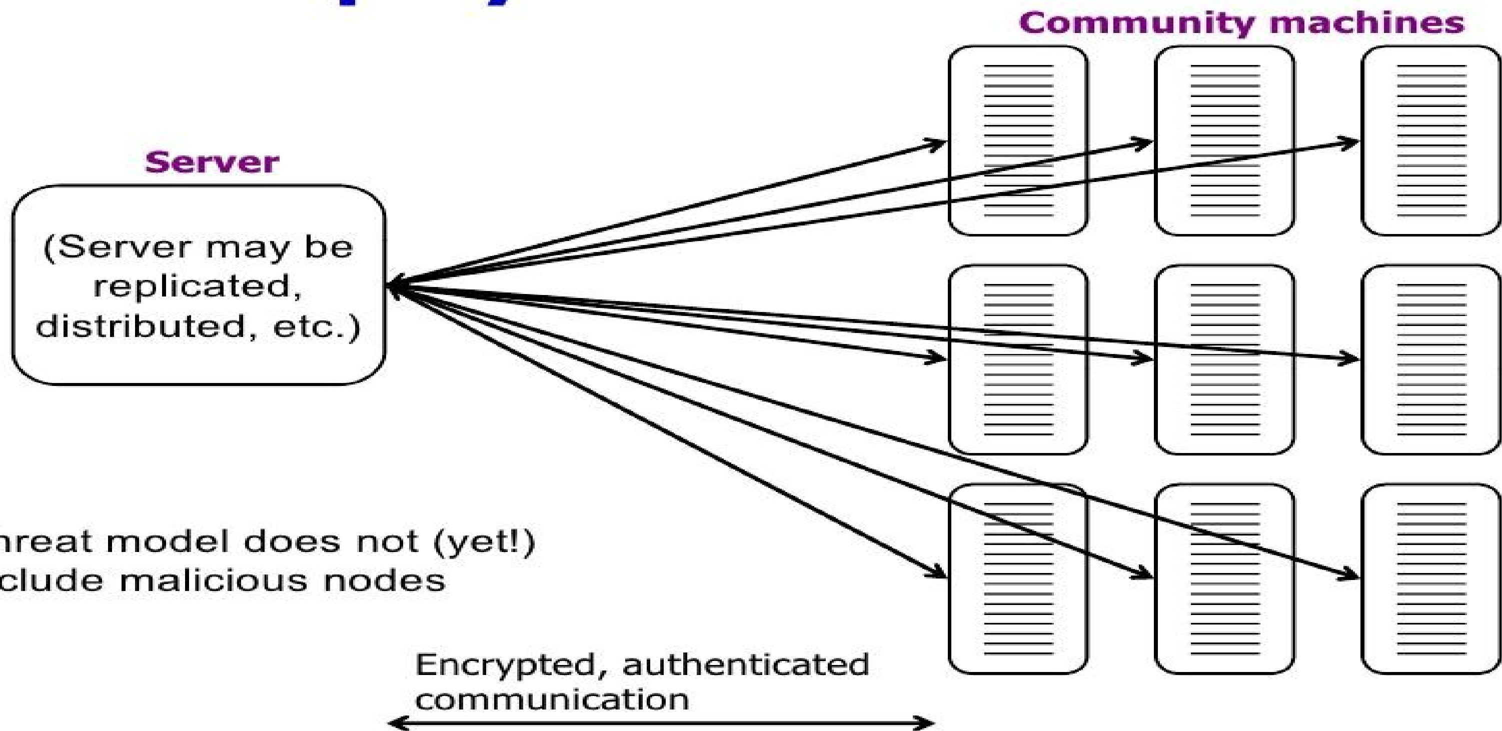
(over a factor of 300 slower)

Patch Creation Time Breakdowns

ClearView took 4.9 minutes to generate a successful patch after an average of 5.4 executions.

Bugzilla Number	Shadow Stack, Heap Guard Runs	Building Invariant Checks	Installing Invariant Checks	Invariant Check Runs	Building Repair Patches	Installing Repair Patches	Unsuccessful Repair Runs	Successful Repair Runs	Total
269095	25.31	12.67[1,0,1]	8.71	51.95(4/28)	10.95[1,0,0]	7.28	51.40(2)	34.5	202.77
*285595	25.38	12.18[0,5,0]	8.47	74.26(6/2216)	11.48[0,3,0]	8.79	-	31.84	172.4
290162	27.14	9.76[2,0,0]	7.79	47.68(2/2)	10.92[1,0,0]	8.4	-	32.64	144.33
295854	32.81	8.82[1,0,0]	9.2	66.29(2/0)	10.34[1,0,0]	8.1	31.11(1)	39.82	206.49
296134	39.31	63.83[0,42,10]	5.89	279.05(?/?)	30.27[0,?,?]	6.23	-	50.22	474.8
!307259	26.14	49.39[0,4,26]	4.45	1235.53(7444/29428)	39.66[0,1,6]	6.28	347.69(7)	-	1709.11
311710a	52	14.22[0,1,2]	9.19	151.29(60/1460)	11.34[0,1,0]	6.83	-	69.05	313.92
311710b	60.48	13.5[0,1,2]	8.27	152.3(60/1460)	13.38[0,1,0]	5.48	-	57.6	311.01
311710c	51.56	17.56[0,1,2]	8.38	161.44(60/1460)	16.17[0,1,0]	8.16	-	64.02	327.29
312278	24.3	8.56[1,0,0]	7.22	48.49(2/0)	11.65[1,0,0]	8	-	33.29	141.51
320182	25.31	12.67[1,0,1]	8.71	51.95(4/28)	10.95[1,0,0]	7.28	51.40(2)	34.5	202.77
*325403	24.21	16.93[0,0,2]	5.9	46.81(4/0)	10.57[0,0,2]	6.01	-	33.48	143.91

A deployment of ClearView



Three sources of inefficiency: -

1. Warming up the Determina Managed Program Execution Environment Code Cache

2. Using Windows Event Queues as the communication mechanism between community members and the centralized servers

3. Compiling the invariant check and repair patches

Limitations:

ClearView is not to correct every conceivable error.

The goal is instead to correct a realistic class of errors to enable applications with high availability requirements to successfully provide service in spite of these errors.

However it might fail to repair the error or it might degrade the application

Discussion 1:

What do you think are the possible sources of inefficiency for ClearView performance

Discussion 2:

What happens if another failure is triggered while ClearView is trying to repair a failure?

Discussion 3:

It is possible for a ClearView repair patch to impair the functionality of the application. The authors provide no formal method to check correctness other than the fact that the software continues execution.

Discussion 4:

The team has performed the experiment in a very controlled environment. How scalable do you think is this to industry level environment.

Discussion 5:

The Red Team exercise was first performed on

Dell 2950 rack-mount machine
16 GB of RAM and two 2.3 GHz Intel Xeon processors, each with four processor cores.
Firefox was run inside VMware virtual machines under ESX servers.
Windows XP Service Pack 2

One exploit (Stack Overflow) did not trigger

But it worked on

1.8 GHz AMD
Opteron machine with four processor cores and 8 Gbytes of RAM
Windows XP Service Pack 2.

How generalized is ClearView?

Discussion 6:

How advantageous is Clearview ? Because industry applications are kept available through redundancy, is clearview a proper alternative to redundant resources?

Discussion 7:

Invariant detection and enforcement in Clearview is localized. How can it patch when program use some global state?

References :

1. http://people.csail.mit.edu/zichaoqi/PatchAnalysis/ClearView.html
2. http://www.thegeekstuff.com/2013/06/buffer-overflow/?utm_source=feedly