ASSURE: Automatic Software Self-Healing Using REscue points

Automatically Patching Errors in Deployed Software (ClearView)

Strategies for resolving vulnerabilities

- Techniques exist for detecting or preventing system compromise
  - Address-space layout randomization
  - Data Execution Prevention
  - Control Flow Integrity
- These techniques crash the program upon detecting the potential compromise of a system
- What to do when an attack is detected?
  - Wait for a patch
  - Restart the system
- An attack may succeed after many failed attempts

Automatically patching vulnerabilities

- Another strategy: Patch binaries so execution can proceed after unanticipated errors occur
  - Detected faults can be aborted while attempting to leave system running in a consistent state
  - Program may exhibit "impossible" behavior after this occurs
- ASSURE: When a fault first occurs, patch the binary to add error handling code so the program can undo the fault continue after such violations happen in the future
- ClearView: When a violation first occurs, use heuristics to patch the binary to prevent the violation in the future

ASSURE and ClearView

ASSURE	ClearView
Applications are profiled before deployment to find rescue points	Trace entries, invariants and control flow graphs are collected at runtime
Applications run with special kernal modules, Dyninst, and Zap for patching, checkpointing, system call virtualization	Applications run under the Determina execution environment
Checkpointing and logging allows a snapshot to be taken when a fault occurs	Stack traces are collected when a fault occurs
Rescue points are selected based on the snapshot and profiling information	Candidate correlated invariants are selected based on the stack trace and invariant information collected at runtime
Patches are tested with automatic and user-supplied test suites	Patches are tested under the system's normal workload and by replaying the error (?)
Patches allow applications to return from the faulting function and continue normally instead of crashing	Patches prevent the application from ever reaching the fault state

ASSURE: Automatic Software Self-healing Using REscue points

- ASSURE searches application code for *rescue points*, which are programmer-defined error handling routines
- When an unanticipated fault is detected during execution, ASSURE patches the binary so execution is redirected to a rescue point when a fault occurs
  - This prevents the application from crashing

Process overview

Profile application before deployment

- Collect rescue point data

Deploy and wait for fault to occur

- Perform snapshots and trace system calls

Select a rescue point

- Create a patch that activates this rescue point

Test the rescue point

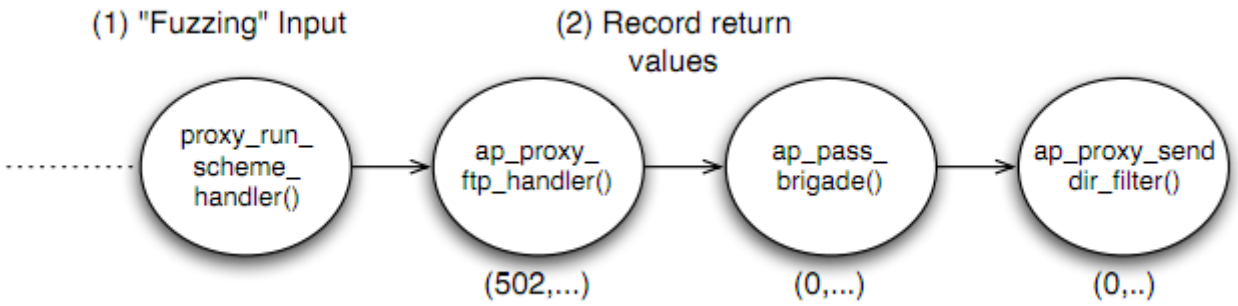
- Restore to the snapshot before the fault to test performance
- Run test cases to find semantic bugs

Deploy the rescue point

- Use Dyninst to deploy the rescue point in a running process

Pre-deployment application profiling

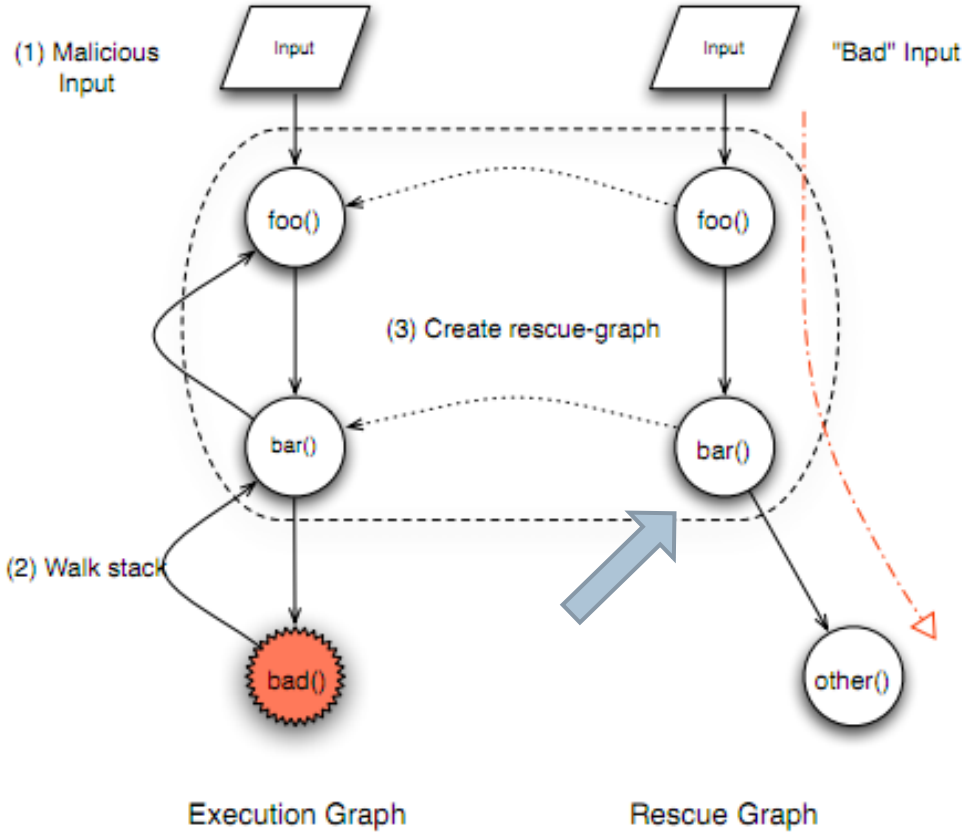
- Before deploying the application, ASSURE profiles it to find rescue points, which often correspond to programmer-defined error handlers
- Bad inputs are generated using fuzzing (random character generation?) and "fault injection".
- Rescue traces are collected in the form of call graphs



Runtime monitoring

- To enable the testing of candidate repairs, ASSURE must checkpoint and log during normal execution
  - Checkpoints are taken periodically
  - System calls are logged between checkpoints, enforcing that a global ordering is maintained between dependent system calls in a multi-process application
- Faults can be detected as segmentation faults or other fault detection tools (stack smashing protectors, etc)

Rescue point selection



- Rescue points are identified through common subgraphs between profiling runs and the faulting run
- Rescue points with a short distance to the faulting function are preferred
- The return value distribution at the rescue point is checked to find the most frequent (non-pointer) return value

Developing a patch to activate the rescue point

- Rescue points are "forked" off through copy-on-write checkpointing
- Multi-process applications are globally checkpointed using the Zap virtual execution environment

```
int rescue_point( int id, fault_t fault ) {
    int rid = rescue_capture(id, fault);
    if (rid < 0)
        handle_error(id);  /* rescue point error */
    else if (rid == 0)
        return get_rescue_return_value(fault);
    /* all ok */
    ...
}
```

Rescue point testing and deployment

- The rescue point is applied to the faulting application at its most recent checkpoint
- Fault recovery is tested
- User-supplied test cases are run to ensure that the resulting application is semantically equivalent
- Rescue points are deployed using runtime binary patching

Evaluation

- Bugs were injected during benchmarks
- ASSURE enabled survivability for each fault

Application	Version	Bug	Reference	Depth	Value	Benchmark
Apache	1.3.31	Buffer overflow	CVE-2004-0940	1	NULL	httperf-0.8
Apache	2.0.59	NULL dereference	ASF Bug 40733	3	502	httperf-0.8
Apache	2.0.54	Off-by-one	CVE-2006-3747	2	-1	httperf-0.8
ISC Bind	8.2.2	Input Validation	CAN-2002-1220	2	-1	dnsperf 1.0.0.1
MySQL	5.0.20	Buffer overflow	CAN-2002-1373	2	1	sql-bench 2.15
Squid	2.4	Input Validation	CVE-2005-3258	1	void	WebStone 2.5b3
OpenLDAP	2.3.39	Design Error	CVE-2008-0658	2	80	DirectoryMark 1.3
PostgreSQL	8.0	Input Validation	CVE-2005-0246	1	0	BenchmarkSQL 2.3.2

Evaluation

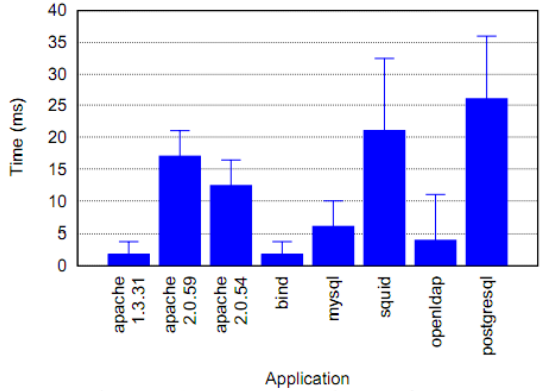


Figure 6: Rescue-point to fault

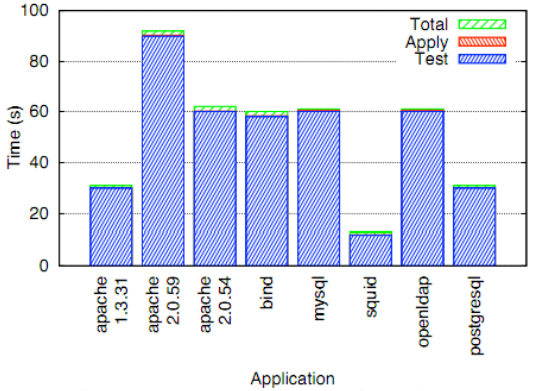


Figure 7: Patch generation time

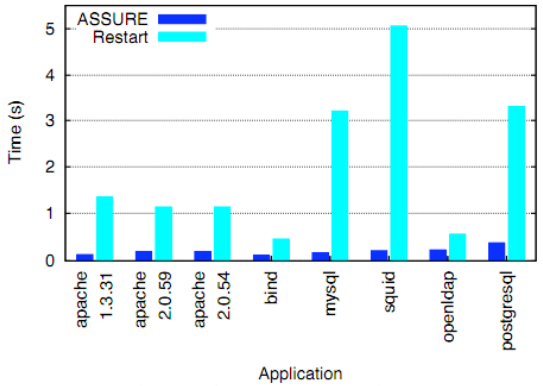


Figure 8: Recovery time

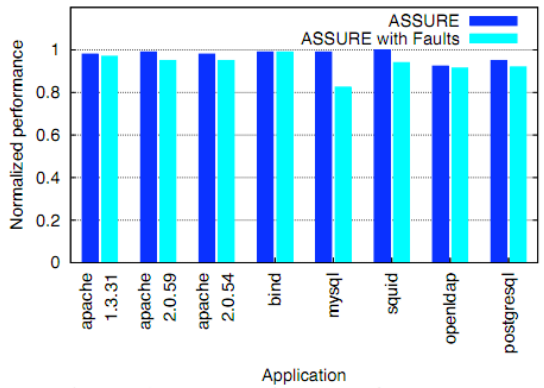


Figure 9: Normalized performance

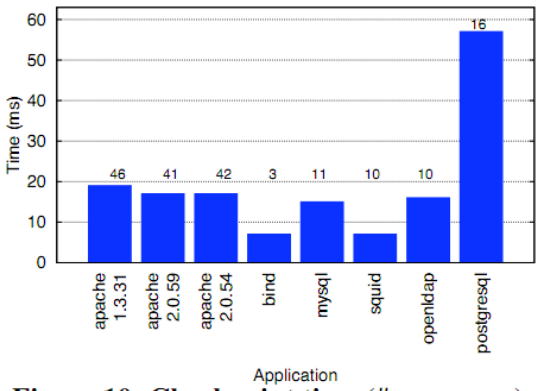


Figure 10: Checkpoint time (# processes)

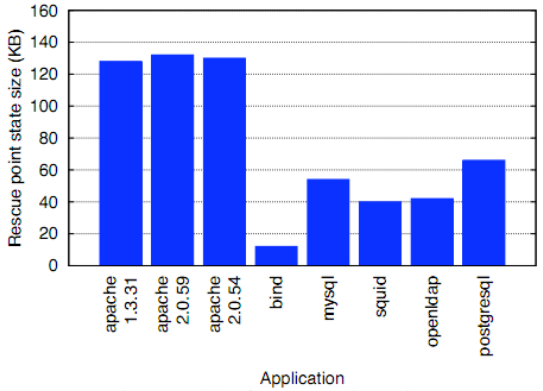


Figure 11: Checkpoint size

Possible limitations

- Doesn't prevent any vulnerabilities from being successfully exploited
  - "[swift patch deployment] allows for the deployment of critical fixes that could curtail the spread of large-scale epidemics such as in the case of worms"?
- Changes to external resources may not be undone upon rollback
- Test cases may depend on external resources
- Rescue points may not correspond to error states

Automatically Patching Errors in Deployed Software

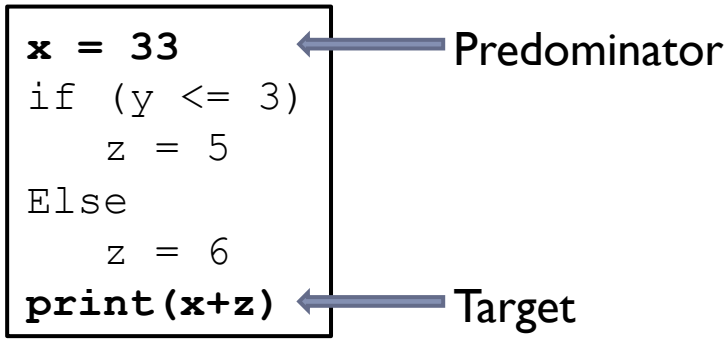
- ClearView is "a system to automatically patch errors in deployed software"
- Programs run under the Determina Managed Program Execution Environment to enable instrumentation and runtime patching
- ClearView detects exploits using heap canaries and Determina detects illegal control flow transfers through program shepherding
  - Program shepherding prevents data execution and requires libraries to be called through exported entry points (Kiriansky, 2002)

Monitoring program execution

- ClearView uses the Daikon invariant learner to infer variants on the program's variables during normal execution
- Daikon dynamically detects *likely invariants* at runtime by observing the values that are assigned to variables
- Some invariants (Ernst, 2007):
  - $x = a$
  - $x \neq 0$
  - $a \leq x \leq b$
  - $x \leq y$
- Clearview instruments assignment instructions using Determina to generate traces for Daikon

Monitoring program execution

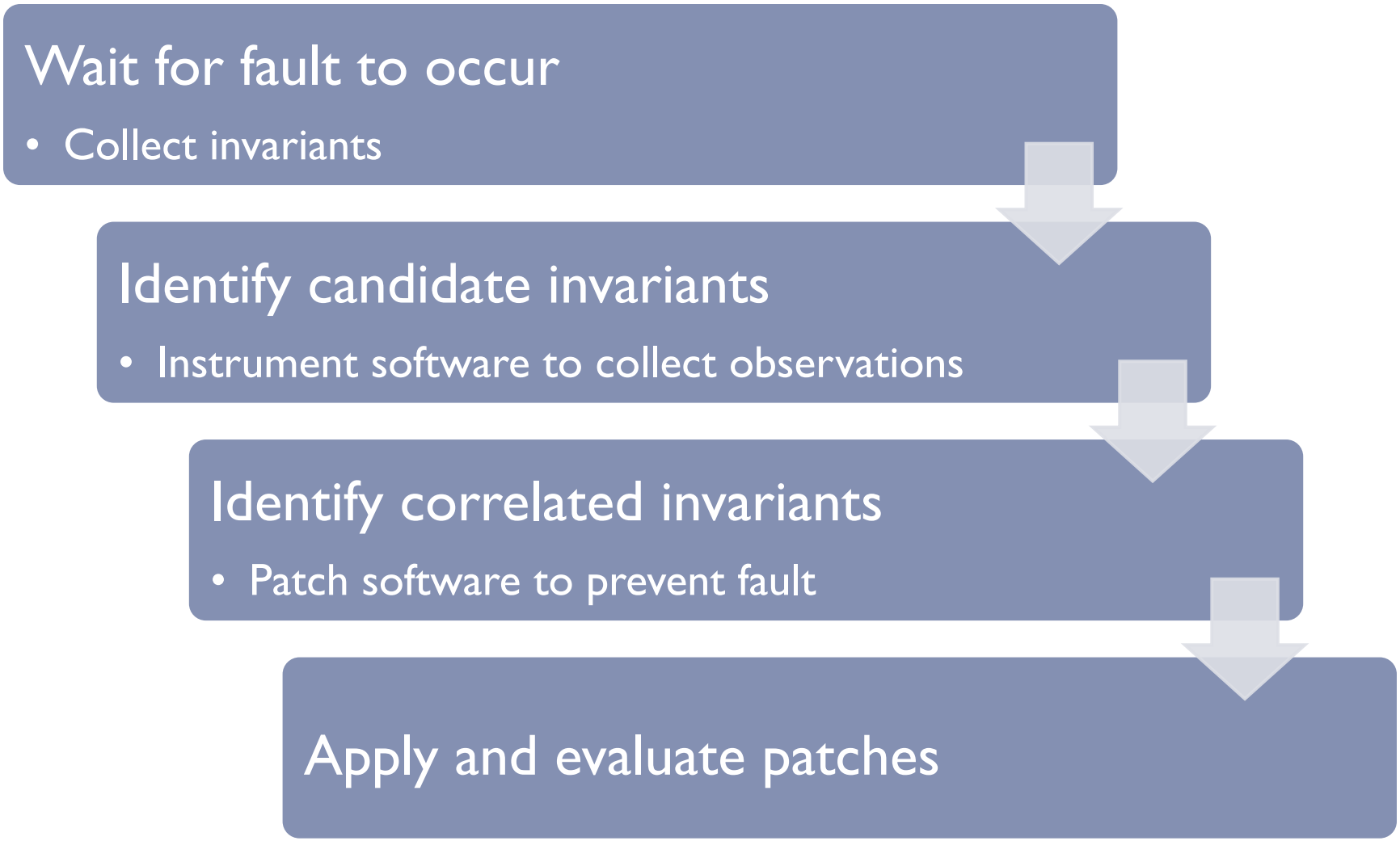
- Invariants are restricted so Daicon only considers a subset of the program's variables at any point
- Invariants are computed considering one instruction at a time (a "target instruction")
- Invariants are restricted to variables computed by the target instruction or a predominator instruction in the same procedure



Monitoring program execurtion

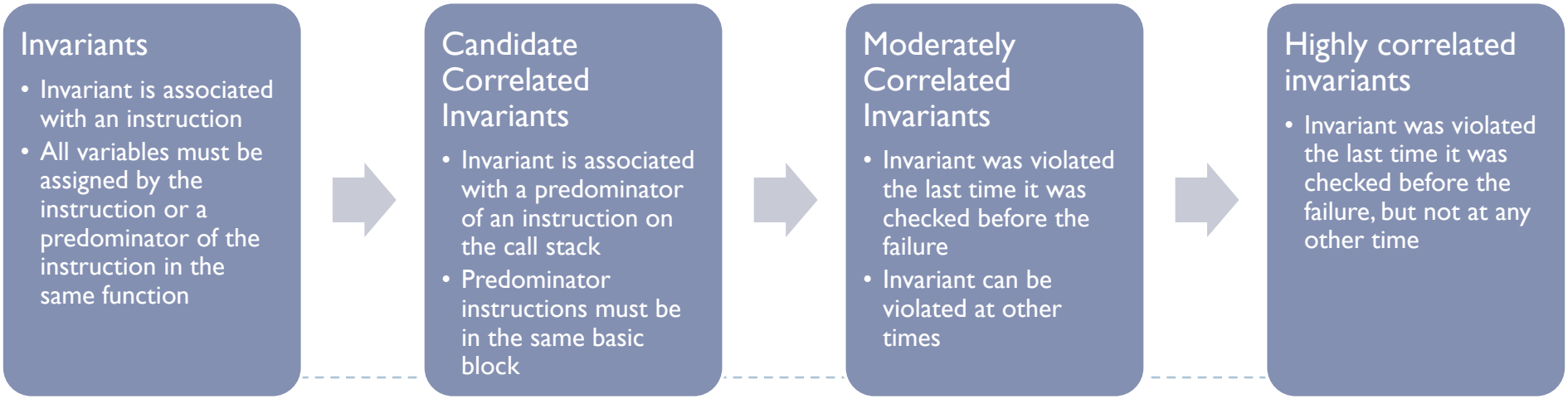
- Procedure control flow graphs are dynamically computed to map basic blocks to procedures and compute predominators
- Trace and control flow data is optimized to find variables which always have the same value, infers the proper position of the stack pointer, and infers which variables are pointers

Process overview



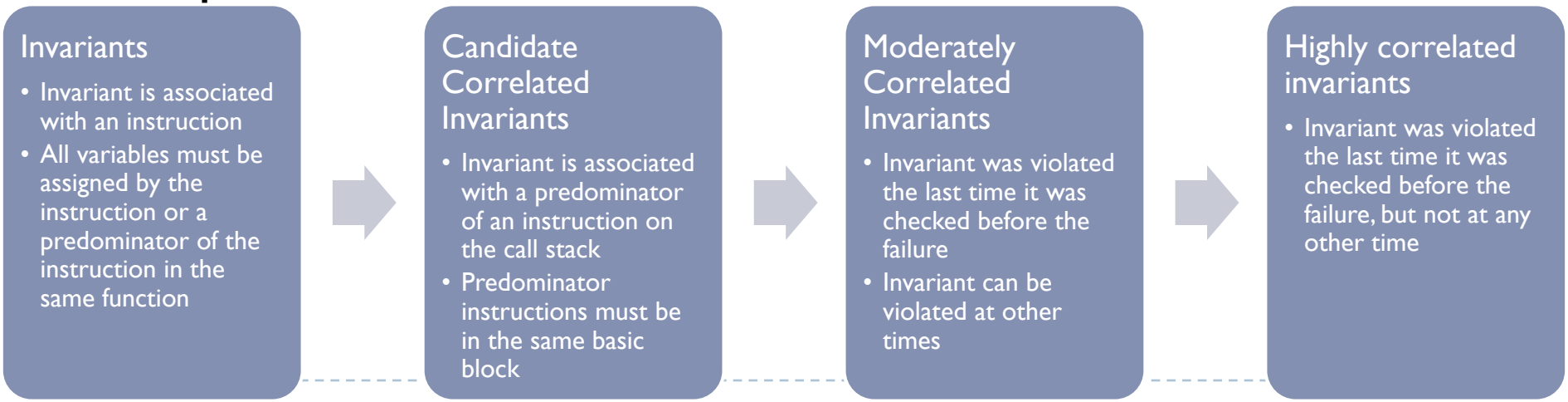
Correlating invariant violations with failure

- When a failure occurs, instructions are identified from the call stack
- *Candidate correlated invariants* are invariants located at these instructions or their predominators
- For optimization, the invariant's instruction must be in the basic block as the instruction from the call stack



Correlating invariant violations with failure

- The program is instrumented to monitor when invariants are satisfied or violated
- These observations are correlated with failures (what happened the last time an invariant was checked before the failure?)
- Highly or moderately correlated invariants are converted to patches



Creating patches

- Instructions are patched to prevent the invariant from being violated
- There may be multiple ways to ensure that a correlated invariant is never violated
  - Variable assignments can be modified to prevent a violation
  - An invariant involving multiple variables could be patched in multiple ways
  - Function calls can be skipped
  - A function return can be forced when the invariant is violated

Evaluating patches

- Repairs are evaluated against the application's normal workload or a replayed attack (?)
- Repairs fail if they do not prevent the fault or if they disrupt the normal functioning of the program
- Repairs succeed if they prevent the failure and the application runs for 10 seconds without crashing
- When a repair fails, other patches are tried to find one that fails less often

Application communities

- Identical deployments of the same application can pool their observations and share patches
  - Learned invariants are uploaded to a central server
  - Candidate invariants are instrumented on all instances of an application to collect correlation data
- Software monoculture as an asset?

Evaluation of ClearView

- ClearView was evaluated on Firefox through a Red Team exercise
- ClearView:
  - Automatically generated patches to fix 7 exploits
  - Generated patches to fix 2 exploits after reconfiguration
  - Failed to patch 1 exploit

Performance evaluation of ClearView

- 300x performance penalty when learning invariants (Invariants were learned before the exercise.)
- Application performance penalty:

ClearView Configuration	Page Load Time (seconds)	Overhead Ratio
Bare Firefox	7.5	1.0
Memory Firewall	11.04	1.47
Memory Firewall + Shadow Stack	14.90	1.97
Memory Firewall + Heap Guard	18.97	2.53
Memory Firewall + Heap Guard + Shadow Stack	22.70	3.03

Performance evaluation of ClearView

- Time required to build and test patches:

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 Run	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)	34.50	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	172.40
290162	27.14	9.76 [2,0,0]	7.79	47.68 (2/2)	10.92 [1,0,0]	8.40	-	32.64	144.33
295854	32.81	8.82 [1,0,0]	9.20	66.29 (2/0)	10.34 [1,0,0]	8.10	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	474.80
!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	52.00	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	13.50 [0,1,2]	8.27	152.30 (60/1460)	13.38 [0,1,0]	5.48	-	57.60	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	24.30	8.56 [1,0,0]	7.22	48.49 (2/0)	11.65 [1,0,0]	8.00	-	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)	34.50	202.77
*325403	24.21	16.93 [0,0,2]	5.90	46.81 (4/0)	10.57 [0,0,2]	6.01	-	33.48	143.91

Possible limitations

- Downselection of invariants and patches causing ClearView to fail to patch a vulnerability
- Functionality impairment
- Patch subversion

ASSURE and ClearView

ASSURE	ClearView
Applications are profiled before deployment to find rescue points	Trace entries, invariants and control flow graphs are collected at runtime
Applications run with special kernal modules with Dyninst for patching and Zap for system call virtualization	Applications run under the Determina execution environment
Checkpointing and logging allows a snapshot to be taken when a fault occurs	Stack traces are collected when a fault occurs
Rescue points are selected based on the snapshot and profiling information	Candidate correlated invariants are selected based on the stack trace and invariant information collected at runtime
Patches are tested with automatic and user-supplied test suites	Patches are tested under the system's normal workload and by replaying the error (?)
Patches allow applications to return from the faulting function and continue normally instead of crashing	Patches prevent the application from ever reaching the fault state