# Automatically Patching Errors in Deployed Software

Jeff H. Perkins[α], Sunghun Kim[β], Sam Larsen[γ], Saman Amarasinghe[α], Jonathan Bachrach[α],
Michael Carbin[α], Carlos Pacheco[δ], Frank Sherwood, Stelios Sidiroglou[α],
Greg Sullivan[ε], Weng-Fai Wong[ζ], Yoav Zibin[η] Michael D. Ernst[θ], and Martin Rinard[α]
[α]MIT CSAIL, [β]HKUST, [γ]VMware, [δ]BCG, [ε]BAE AIT, [ζ]NUS, [η]Come2Play, [θ]U. of Washington
jhp@csail.mit.edu, mernst@cs.washington.edu, rinard@csail.mit.edu

## ABSTRACT

We present ClearView, a system for automatically patching errors in deployed software. ClearView works on stripped Windows x86 binaries without any need for source code, debugging information, or other external information, and without human intervention.

ClearView (1) observes normal executions to learn invariants that characterize the application's normal behavior, (2) uses error detectors to distinguish normal executions from erroneous executions, (3) identifies violations of learned invariants that occur during erroneous executions, (4) generates candidate repair patches that enforce selected invariants by changing the state or flow of control to make the invariant true, and (5) observes the continued execution of patched applications to select the most successful patch.

ClearView is designed to correct errors in software with high availability requirements. Aspects of ClearView that make it particularly appropriate for this context include its ability to generate patches without human intervention, apply and remove patches to and from running applications without requiring restarts or otherwise perturbing the execution, and identify and discard ineffective or damaging patches by evaluating the continued behavior of patched applications.

ClearView was evaluated in a Red Team exercise designed to test its ability to successfully survive attacks that exploit security vulnerabilities. A hostile external Red Team developed ten code injection exploits and used these exploits to repeatedly attack an application protected by ClearView. ClearView detected and blocked all of the attacks. For seven of the ten exploits, ClearView automatically generated patches that corrected the error, enabling the application to survive the attacks and continue on to successfully process subsequent inputs. Finally, the Red Team attempted to make ClearView apply an undesirable patch, but ClearView's patch evaluation mechanism enabled ClearView to identify and discard both ineffective patches and damaging patches.

## 1. Introduction

We present ClearView, a system for automatically correcting errors in deployed software systems with high availability requirements. Previous research has shown how to detect errors, for example by monitoring the execution for buffer overruns, illegal control transfers, or other potentially incorrect behavior [19, 31, 21]. The standard mitigation strategy is to terminate the application, essentially converting all errors into denial of service. In many important scenarios, system availability is a strict requirement. In such scenarios, it is imperative to eliminate the denial of service and enable the application to continue to provide service even in the face of errors.

ClearView can automatically correct previously unknown errors in commercial off-the-shelf (COTS) software systems. It patches running applications without requiring restarts or otherwise perturbing the execution. It requires no human interaction or intervention. And it works on stripped Windows x86 binaries without access to source code or debugging information.

Figure 1 presents the architecture of ClearView, which has five components:

- **Learning:** ClearView dynamically observes the application's behavior during normal executions to infer a model that characterizes those normal executions. The model is a collection of properties, also called invariants, over the observed values of registers and memory locations. As the name suggests, invariants are always satisfied during the observed normal executions. As ClearView observes more executions, its model becomes more accurate. Our current ClearView implementation uses an enhanced version of Daikon [14] as its learning component.
- **Monitoring:** ClearView monitors detect failures to classify each execution as normal or erroneous. For each erroneous execution, the monitor also indicates the location in the binary where it detected the failure. ClearView is designed to incorporate arbitrary monitors. Our current implementation uses two monitors: Heap Guard (which detects out of bounds memory writes) and Determina Memory Firewall (a commercial implementation of program shepherding [21] which detects illegal control flow transfers). Each monitor prevents negative consequences by terminating the application when it detects a failure. Our current monitors have no false positives in that they never classify a normal execution as erroneous. But they are also only designed to detect a specific class of errors (heap buffer overflows and illegal control flow transfers). ClearView is not designed to eliminate all failures, only those that a monitor detects.
- **Correlated Invariant Identification:** When a monitor detects a failure, ClearView applies a set of patches that check previously learned invariants close to the location of the failure. Note that these invariant checking patches are not intended to correct errors or eliminate the failure. The goal is instead to find a set of *correlated invariants* that characterize normal and erroneous executions. Specifically, each correlated invariant is always satisfied during normal executions but violated during erroneous executions (which typically occur in response to repeated or replayed attacks).
- **Candidate Repair Generation:** For each correlated invariant, ClearView generates a set of candidate repair patches that enforce the invariant. Some of these patches change the values of registers and memory locations to reestablish the invariant whenever it is violated. Others change the flow of control to enforce observed control flow invariants.

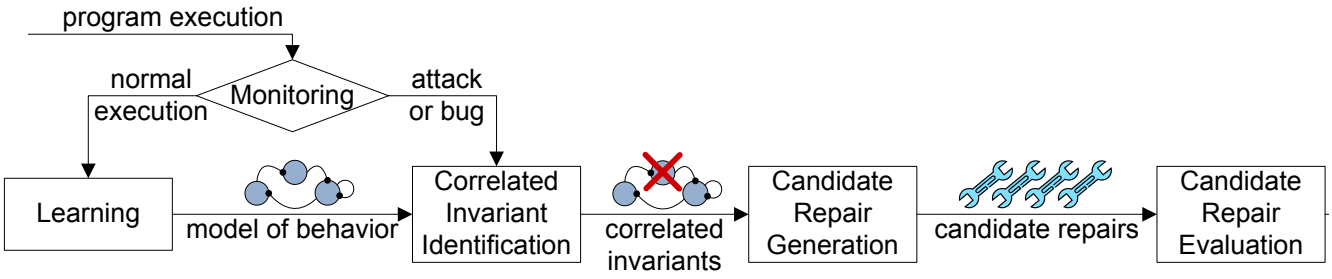


**Figure 1: The ClearView Architecture**

The hypothesis is that some errors violate invariants, that enforcing violated invariants can correct the effects of these errors, and that correcting these effects can, in turn, change the execution of the application to eliminate the corresponding failure. The goal is to find a patch that corrects the execution after the first error has occurred but before any effects have propagated far enough to make a failure inevitable. To accomplish this goal, ClearView must find and correct errors that occur early enough in the execution to make the execution salvageable via the invariant enforcement mechanism.

- **Candidate Repair Evaluation:** A candidate repair patch may have no effect or even a negative effect on the patched application. ClearView therefore evaluates patches by continuously observing patched applications as they execute. It ranks each patch based on whether it observes any failures or crashes when the patch is in place. At each point in time ClearView attempts to minimize the likelihood of negative effects by applying the most highly ranked patch. The fact that patches affect the execution only when a correlated invariant is violated also tends to minimize the possibility that they will negatively affect normal executions.

  We acknowledge that, in some cases, the application's maintainers (if they exist) may wish to find and eliminate the defect in the source code (if it still exists) responsible for the failure. ClearView supports this activity by providing information about the failure, specifically the location where it detected the failure, the correlated invariants, the strategy that each candidate repair patch used to enforce the invariant, and information about the effectiveness of each patch. This information may help the maintainers more quickly understand and eliminate the corresponding defect. In the meantime, the automatically generated ClearView patches can enable the application to survive to provide acceptable service.

ClearView is designed around the concept of learning from failure (and from success, too). This process of learning enables the quality of ClearView's patches to improve over time, similarly to a biological immune system. The first time it detects a failure, ClearView learns a failure location in the application. This information enables ClearView to target subsequent instrumentation and intervention only where it is likely to be effective. The next several times it encounters the failure, ClearView learns the variables and data structures that are corrupted and the invariants that are violated. This information enables ClearView to generate candidate repair patches that may correct the error and eliminate the failure. Then, ClearView applies the patches. As the application processes subsequent inputs, ClearView obtains information that it can use to evaluate the effectiveness of each patch. This evaluation enables ClearView to discard ineffective or damaging patches while applying successful patches that are able to eliminate the failure without negatively affecting the application.

## 1.1 Red Team Evaluation

As part of DARPA's Application Communities program (`www.darpa.mil/IPTO/programs/ac/ac.asp`), DARPA hired Sparta, Inc. (`www.sparta.com`) to perform an independent and adversarial Red Team evaluation of ClearView. The goal was to evaluate ClearView's effectiveness in eliminating security vulnerabilities. Given the need for fast automated response to attacks that target such vulnerabilities, the time lag typically associated with human intervention (it takes 28 days on average for maintainers to develop and distribute fixes for such errors [41]), and ClearView's ability to quickly and automatically generate and evaluate patches without human intervention, we anticipate that ClearView may be especially useful in this context.

In the Red Team exercise, ClearView protected an application community, which is a set of computers that all run the same software — in this case, the Firefox web browser. The community cooperates to learn information about attacks, then uses that information to provide immunity to all members (including members with no previous exposure to the attack) after only several have been attacked. The community also reduces the time required to obtain an effective patch.

During the Red Team exercise, the Red Team developed ten binary code injection exploits and used these exploits to repeatedly attack the community. The results show that:

- **Attacks Blocked:** ClearView detected and blocked all of the attacks.
- **Successful Continued Execution:** For seven of the ten exploits, ClearView automatically generated and applied a patch that corrected the underlying error and enabled the application to execute successfully even in the face of the attack. For two of the remaining three exploits, changes to the ClearView Red Team exercise configuration enabled ClearView to automatically generate similarly successful patches.
- **Patch Quality:** Some of the candidate repair patches that ClearView evaluated either did not eliminate the failure or introduced new negative effects such as causing the patched application to crash. ClearView's patch evaluation mechanism recognized and discarded such patches, enabling ClearView to find and distribute patches that eliminated the failure without negative effects. The Red Team was unable to find a legitimate input that the final patched version of Firefox processed incorrectly.
- **No False Positives:** The Red Team was unable to elicit any ClearView false positives. Specifically, the Red Team was unable to cause ClearView to apply patches in the absence of an attack.

By making it possible to automatically survive otherwise fatal errors and attacks, our techniques may increase the robustness and availability of our computing infrastructure in a world increasingly full of errors and security vulnerabilities.

## 1.2 Contributions

This paper makes the following contributions:

- **Learning:** It shows how to observe normal executions to automatically learn invariants in stripped Windows x86 binaries with no access to source code or debugging information.
- **Invariants and Patches:** It identifies a set of invariants and corresponding patches that can automatically detect and correct errors and security vulnerabilities.
- **Invariant Selection and Patch Evaluation:** It shows how to automatically respond to a failure by selecting a promising set of correlated invariants to enforce, then evaluating the corresponding candidate repair patches to find a patch that enables the application to survive errors and attacks without service interruptions. Because the patch evaluation mechanism continually evaluates the effectiveness of the patches, it can recognize and discard ineffective or damaging patches even long after they were originally applied.
- **Application Communities:** It shows how to enable communities of machines to work together as a group to survive errors and attacks and continue to successfully execute. The members of the community share invariant learning information and benefit from the experience of other members to obtain full immunity without prior exposure.
- **Red Team Evaluation:** It presents experimental results from a hostile Red Team evaluation of ClearView. These results show that, for seven of the ten security exploits that the Red Team developed, ClearView produced patches that enabled the application to survive the attacks and continue on to successfully process subsequent inputs. There were no false positives — the Red Team was unable to make ClearView apply a patch in the absence of an error or attack.
- **Natural Resilience:** It provides further evidence that, when augmented with simple survival techniques that enable continued execution through errors without termination, large software systems are naturally resilient to errors and attacks. This empirically observed resilience suggests that the most productive way to achieve robust software systems is to automatically apply survival techniques that modify the application and/or the execution environment to enable continued execution in the face of errors. This approach stands in stark contrast to standard attempts to develop error-free systems on top of existing brittle programming languages and execution environments. It also provides additional intriguing evidence that the complexity inherently present in large software systems may protect such systems from any negative effects of the localized perturbations characteristic of errors after the automatic application of survival techniques.

# 2. ClearView Implementation

This section describes ClearView's implementation of each component of the architecture of Figure 1: learning (Section 2.2), monitoring (Section 2.3), correlated invariant identification (Section 2.4), repair generation (Section 2.5), and repair evaluation (Section 2.6). All of these components build on the capabilities of various components of the Determina commercial product suite (Section 2.1).

We adopt the following conceptual framework from the software reliability community. A *defect* is a mistake in the source code of the application. An *error* occurs when the application does something incorrect, such as compute an incorrect value, while it is running. In general, a *failure* is an observable error, i.e., a violation of the application's specification that is visible to a user. In this paper we narrow the definition of failure to include only errors that are detected by a ClearView monitor. Other errors that cause the application to terminate are *crashes*. The remaining kinds of errors are *anomalies*, i.e., violations of the application's specification that do not cause the application to terminate and are not detected by a ClearView monitor. An *exploit* is an input to an application that causes the application to violate its specification. An *attack* is a presentation of an exploit to an application. Note that defects and errors are undesirable primarily to the extent that they cause failures, crashes, or anomalies.

## 2.1 Determina Components

The Determina Managed Program Execution Environment uses DynamoRIO [4] to enable the efficient, transparent, and comprehensive manipulation of an unmodified application at the binary code level with reasonable time and memory overhead. All code executes out of a code cache, with an exported interface that allows plugins to validate and (if desired) transform new code blocks before they enter the cache for execution. Plugins can also eject previously inserted code blocks from the cache. Plugins can use this functionality to apply and remove patches to and from running applications without otherwise perturbing the execution. ClearView uses this capability to apply and remove patches that enable ClearView to check and enforce invariants.

Determina Memory Firewall detects all illegal control flow transfers and intervenes to terminate the execution before any injected code can execute (note that the detection and intervention typically take place only after the attack has corrupted the application state to the point where the application can no longer execute successfully). Specifically, when an instruction in the code cache attempts to jump to code outside of the cache, Memory Firewall performs a validation check on the control flow transfer [21]. If the transfer passes the validation check, the Managed Program Execution Environment links the target code block into the code cache (after applying any desired instrumentation or patches), then jumps to this target code block to continue the execution from the code cache. This implementation of program shepherding protects client applications from binary code injection attacks [21].

The Determina Management Console can monitor and control a large set of distributed client machines. It runs on a central server that stores patches and communicates securely with instantiations of the Determina Node Manager running on each client. Each Node Manager interacts with its corresponding Managed Program Execution Environment instance to appropriately apply and remove patches to and from running and newly launched applications. Several ClearView Management Console plugins build on the Management Console functionality to coordinate ClearView's interaction with client machines so that they appropriately apply and remove ClearView patches.

## 2.2 Learning

The learning component observes normal executions to infer a model of the normal behavior of the application. This model consists of a set of invariants that were satisfied in all observed executions. Each invariant is a logical formula that was always satisfied at a specific instruction in the application. Because ClearView operates on binaries, the variables in the formulas represent values (specifically, the values of registers and memory locations) that are meaningful at the level of the compiled binary.

### *2.2.1 Daikon*

ClearView uses Daikon [15] as its learning component. A Daikon implementation has two components: a front end that extracts trace data from a running application, and a core inference engine that processes the trace data to infer the invariants. Daikon was orig-

inally developed to learn source-level invariants. As part of the ClearView development effort we used the Determina Managed Program Execution Environment to implement a new Daikon front end (the x86 front end) that instruments the instructions in basic blocks, as they enter the code cache, to emit the appropriate trace data when they execute.

For each instruction, the trace data includes the values of all operands that the instruction reads and all addresses that the instruction computes. Consider, for example, `mov [ebp+12], eax`, which moves the value in memory location `ebp+12` into the register `eax`. This instruction computes one address (`ebp+12`) and reads one operand (the value in this address). Every time the instruction executes, the ClearView instrumentation produces a trace entry containing this data.

### 2.2.2 Invariant Variables and Locations

Several considerations determine the set of variables over which the core Daikon inference engine operates when it infers the invariants for a given instruction. First, the set of variables must be large enough to enable the inference of meaningful invariants whose enforcement can correct our target class of errors. Second, the set must be small enough to make the inference task computationally tractable. Finally, the values in the variables must be defined in all possible executions (and not just the observed executions).

At each instruction (call this instruction the *target instruction*), the core Daikon inference engine operates over all variables that are computed in the procedure containing the target instruction (see Section 2.2.3) by either the target instruction itself or another instruction that predominates the target instruction.[1] Each invariant must also include a variable that the target instruction computes.

### 2.2.3 Procedure Control Flow Graphs

To compute predominators, ClearView builds a control flow graph for each executed procedure. The nodes in the graph are basic blocks (as determined by the Determina Managed Program Execution Engine as it executes the application) instead of instructions. The edges represent the flow of control between basic blocks.

The control flow graph construction algorithm uses a novel combined static and dynamic analysis. This analysis eliminates the need to find procedure entry points statically (a complex task in a stripped x86 executable). It maintains a database of known control flow graphs (with one control flow graph for each dynamically encountered procedure). It finds new procedures by considering each basic block the first time it executes. If the basic block is not already in a known control flow graph, the algorithm assumes that the basic block is the entry point for a new procedure. It then uses symbolic execution to dynamically trace out the basic blocks in and construct the control flow graph for this new procedure. It uses the first basic block as the entry basic block of the new procedure.

During the symbolic execution, the algorithm ends the procedure at return instructions and indirect jump instructions for which it cannot compute the jump target. Note that this algorithm may break a single static procedure up into multiple dynamically discovered procedures. In practice such procedure fission happens relatively rarely. The only potential drawback is that splitting procedures in this way may reduce the set of values available to the invariant inference engine at a given instruction, which may, in turn, reduce the set of invariants that the inference engine can infer for that instruction.

[1] An instruction `i` predominates an instruction `j` if all control flow paths to `j` must first go through `i`. If `i` predominates `j`, then whenever the flow of control reaches `j`, `i` has previously executed and all of the values computed in `i` are valid.

### 2.2.4 Additional Properties and Optimizations

ClearView analyzes the control flow graphs to identify sets of distinct variables (these variables denote values in registers or memory locations) in the same procedure that always have the same value. It then postprocesses the trace data to remove all occurrences of such variables except the one from the earliest instruction to execute. This optimization reduced the number of inferred invariants by a factor of two, which in turn reduced the learning, invariant checking, and repair evaluation time.

Another ClearView postprocessing step analyzes the trace data to discover invariants involving stack pointer offsets. As a procedure allocates and deallocates local variables and calls other procedures, it may change the value of the stack pointer. Stack pointer offset invariants of the form $\mathtt{sp}_1 = \mathtt{sp}_2 + \mathtt{c}$ capture the resulting relationships between the stack pointer $\mathtt{sp}_1$ at the entry point of the procedure and stack pointers $\mathtt{sp}_2$ at various points within the procedure. ClearView uses this information to adjust the stack pointer appropriately for repairs that skip procedure calls or return immediately from the enclosing procedure (see Section 2.5.1).

We also extended the set of Daikon invariants to include information about which variables contain pointer values. Specifically, if a negative value or value between 1 and 100,000 ever appears in a variable, Daikon infers that it is not a pointer. Otherwise, Daikon infers that it is a pointer. Daikon uses this information to skip the inference of lower-bound or less-than invariants that involve pointers. This optimization reduces the learning, invariant checking, and repair evaluation time.

## 2.3 Monitoring

ClearView can incorporate any monitor that can detect a failure and provide a failure location (the program counter of the instruction where the monitor detected the failure). The current implementation uses Memory Firewall to detect illegal control flow transfer errors. This monitor is always enabled in a running application.

**Heap Guard:** The Heap Guard monitor detects out of bounds memory accesses. It places canary values at the boundaries of allocated memory blocks and instruments all writes into the heap to check if the written location contains the canary value. The presence of the canary value indicates either an out of bounds write or a legitimate previous write (by the application) of the canary value within the bounds of an allocated memory block. When Heap Guard encounters a canary value, it therefore searches an allocation map to determine if the written address is within the bounds of some allocated memory block. If so, normal execution continues; if not, Heap Guard has detected an out of bounds write error. By design, Heap Guard suffers no false positives. It may, however, miss an out of bounds write error if the out of bounds write skips over the boundaries of the allocated memory block.

Heap Guard is useful in two different ways. First, it can detect out of bounds writes that do not eventually cause illegal control flow transfers. In this way, Heap Guard may enable ClearView to detect (and therefore potentially correct) out of bounds write errors that Memory Firewall does not detect.

Second, even when an out of bounds write error would eventually cause an illegal control flow transfer, Heap Guard may detect an earlier error than Memory Firewall would. This earlier detection may enhance ClearView's ability to find a successful patch.

It is possible to dynamically enable and disable Heap Guard as the application executes without otherwise perturbing the execution. ClearView could, for example, use this functionality to run the application without Heap Guard during normal production execution, then turn Heap Guard on when an event (such as a failure) indicates an elevated risk of an out of bounds write error.

**Shadow Stack:** ClearView can optionally maintain an auxiliary shadow procedure call stack. This Shadow Stack enables ClearView to traverse the call stack to find additional candidate correlated invariants in callers of the procedure containing the failure location. Enforcing one of these additional invariants may be critical in enabling ClearView to correct the error.

There are two reasons that ClearView uses the Shadow Stack rather than attempting to unwind the native call stack. First, a variety of optimizations (such as frame pointer removal and heavily optimized caller/callee interactions) can make it difficult to reliably traverse a native call stack. Second, errors (such as buffer overflows) may corrupt the native stack, making it unavailable to ClearView when a monitor detects a failure.

The Shadow Stack contains the addresses of the procedures on the actual stack, with call and return instructions instrumented to maintain the Shadow Stack. The instrumentation is largely performed in-line for efficiency and can be enabled and disabled as the application runs without perturbing the execution.

## 2.4 Correlated Invariant Identification

Given a failure location, ClearView attempts to identify invariants whose violation is correlated with the failure. These *correlated invariants* have two important properties. First, they are always satisfied in normal executions but violated in erroneous executions. Second, they are violated before the failure occurs. The rationale is that using a correlated invariant to force the application back into its normal operating envelope may correct the error, eliminate the failure, and enable the application to continue to operate successfully.

### *2.4.1 Candidate Correlated Invariants*

ClearView uses the procedure call stack to obtain a set of candidate correlated invariants. Specifically, assume that a procedure `P` at instruction `i` is on the call stack when a monitor detects a failure. Then any invariant at a predominator of `i` in `P` is in the candidate correlated invariant set.

Note, however, that ClearView implements an optimization that restricts the set of candidate correlated invariants. Specifically, if an invariant relates the values of two variables, the invariant's instruction must occur in `i`'s basic block for ClearView to include the invariant in the set of candidate correlated invariants. This additional restriction substantially reduces both the invariant checking overhead (see Section 2.4.2) and the number of candidate repairs that ClearView generates and evaluates when a monitor detects a failure (see Section 2.5). In practice this optimization did not remove any useful repairs and (by reducing the number of repairs to evaluate) decreased the amount of time required to find a successful repair.

More generally, ClearView can use virtually any strategy that identifies a set of candidate correlated invariants that is likely to produce a successful repair. The three key considerations are 1) working with the available failure information (in the current implementation this information includes the location of the failure and, if enabled, the Shadow Stack), 2) making the set large enough to include an invariant that produces a successful repair, and 3) limiting the size of the set to make it feasible to efficiently check the invariants (as described below in Section 2.4.2) and evaluate the resulting set of candidate repairs (as described below in Section 2.6). In particular, if the Shadow Stack is not available, ClearView can simply work with invariants associated with instructions close to the failure location. It is also possible to develop strategies that learn clusters of basic blocks that tend to execute together, then work with sets of invariants from clusters containing the basic block where the failure occurred.

### *2.4.2 Checking Candidate Correlated Invariants*

Given a set of candidate correlated invariants, ClearView generates and deploys a set of patches that check each candidate correlated invariant to determine if it is satisfied or violated. Patches that check the value of a single variable execute when the program counter reaches the instruction associated with the variable. Patches that check the relationship between the values of two variables execute when the program counter reaches the second instruction (to execute) of the two instructions associated with the two variables. An auxiliary patch associated with the first instruction stores the value of the first variable for later retrieval when the invariant is checked at the second instruction.

Each invariant checking patch produces an observation every time it executes. Each observation identifies the invariant and the location of the failure that triggered the deployment of the patch. It also indicates whether the invariant was satisfied or violated. In this way the patched application produces, for each combination of invariant and failure location, a sequence of observations.

### *2.4.3 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:** An invariant is highly correlated with the failure if, each time a monitor detected the failure, it was violated the last time it was checked and satisfied all other times it was checked.
- **Moderately Correlated:** An invariant is moderately correlated with the failure if, each time the monitor detected the failure, it was violated the last time it was checked and, at least one of the times the monitor detected the failure, it was also violated at least one other time it was checked.
- **Slightly Correlated:** An invariant is slightly correlated with the failure if, at least one of the times the monitor detected the failure, the invariant was violated at least one of the times it was checked.
- **Not Correlated:** An invariant is not correlated with the failure if it was always satisfied.

Note that correlated invariants need not be violated every time they are checked — one scenario, for example, is that only the last part of an erroneous execution may exhibit erroneous behavior, with the initial parts exhibiting normal behavior during which correlated invariants may well be satisfied. The candidate repair generation phase described below in Section 2.5 uses the classification to select invariants to enforce.

## 2.5 Candidate Repair Generation

Given a set of correlated invariants, ClearView generates a set of candidate repairs to evaluate. Each candidate repair corresponds to a correlated invariant. The patch that implements the repair first checks to see if the invariant is violated. If so, the patch enforces the invariant by changing the flow of control, the values of registers, and/or the values of memory locations to make the invariant true. Patches for invariants involving only a single variable check and enforce the invariant at the variable's instruction. Patches for invariants involving multiple variables check and enforce the invariant at the latest (to execute) of the corresponding instructions for the involved variables. In general, there may be multiple ways to enforce a single invariant. ClearView generates a candidate repair for each such invariant enforcement option.

If there are highly correlated invariants, the current ClearView implementation generates candidate repairs for only those invari-

ants. If there are no highly correlated invariants, ClearView generates candidate repairs only for moderately correlated invariants (if any exist). It would also be possible to generalize this approach to have ClearView generate candidate repairs for all correlated invariants, with the ClearView candidate repair evaluation (see Section 2.6) using the correlated invariant classification to prioritize the evaluation of the corresponding repairs. We next describe the three ClearView invariants and corresponding repairs that were used during the Red Team exercise (see Section 4).

### 2.5.1 One-of invariant

A one-of invariant has the form $v \in \{c1, c2, \ldots, cn\}$, where the `ci` are constants and `v` is a variable or expression. This property identifies all of the values that the variable `v` ever took on at run time. There are `n` repairs of the following form, one for each observed value:

```
if ! (v == c1 || v == c2 || ... || v == cn) then v = ci
```

If the application uses `v` as a function pointer (i.e., `v` is the target of a call instruction), another repair simply skips the call. The repair replaces `call *v` with

```
if (v == c1 || v == c2 || ... || v == cn) then call *v
```

Note that the repair skips the call if the invariant is violated.

A third repair returns immediately from the enclosing procedure (the actual patch also adjusts the stack pointer to remove the arguments to the call and performs other cleanup):

```
if ! (v == c1 || v == c2 || ... || v == cn) then return
```

This repair can be used for any invariant, but ClearView currently uses it only for one-of invariants.

**Rationale:** One-of invariants often characterize the observed targets of function calls that use function pointers. The use of uninitialized memory or incorrect type casts can produce erroneous function pointers. Many security attacks also exploit vulnerabilities that enable attackers to create malicious function pointers. Enforcing the invariant eliminates any illegal control flow transfer, which may, in turn, enable the application to survive the error or attack.

### 2.5.2 Lower-bound Invariant

A lower-bound invariant has the form $c \leq v$, where `c` is a constant and `v` is a variable or expression. One repair has the form:

```
if ! (c <= v) then v = c
```

**Rationale:** One class of defects can cause an array or buffer index to be negative, which can cause the application to read and/or write addresses below the start of the array or buffer. A related class of defects can cause the application to pass a negative number as a length to a procedure such as `memcpy`, which treats the number as a large unsigned integer. The resulting memory copy then writes beyond the end of the buffer.

Such defects typically result in errors that violate a lower-bound invariant such as $0 \leq v$. Enforcing the invariant redirects the out-of-bound index back into the buffer or array, which can prevent memory corruption and enable the application to survive the error.

### 2.5.3 Less-than Invariant

A less-than invariant $v1 \leq v2$ relates two variables or expressions (by contrast, a lower-bound invariant relates a variable and a constant). Less-than invariants can be repaired by adjusting either `v2` (as in the lower-bound repair) or `v1`. One repair is of the form:

```
if ! (v1 <= v2) then v1 = v2
```

**Rationale:** A defect can cause an array or buffer index to exceed the upper bound of the array or buffer, which can cause the application to access addresses above the end of the array or buffer. Such defects typically cause the application to violate a less-than invariant that captures the requirement that the index must be below the upper bound of the array or buffer. Enforcing the invariant redirects the out-of-bound index back into the array or buffer, which can eliminate memory corruption and enable the application to survive the error.

## 2.6 Candidate Repair Evaluation

ClearView evaluates each candidate repair to determine which, if any, is the most effective at correcting the error and enabling the application to operate normally. ClearView considers the repair to have failed if the failure still occurs, a new failure occurs, or the application crashes after repair. ClearView (tentatively) considers the repair to have succeeded if the application has executed with the repair in place for at least ten seconds without failing or crashing.

The repair evaluation is based on relative repair scores. When a repair succeeds, its score increases. When a repair fails, its score decreases. Since the goal is to find a repair that always works, the scoring system is designed to reward repairs that are *always* successful. If a repair ever fails, the system continues to search for a more successful repair. ClearView therefore uses the scoring formula $(s - f) + b$, where $s$ is the number of successes, $f$ is the number of failures, and $b$ is a positive bonus given to any repair that has not yet failed. ClearView uses the following criteria to break ties among repairs with the same score (for instance, all repairs that have never been tried have score $b$):

- **Earlier Repairs First:** ClearView prefers repairs that take effect earlier in the execution. In a given basic block or procedure, ClearView prefers repairs from earlier instructions. Among repairs in different procedures, ClearView prefers repairs in procedures lower on the call stack. The rationale is to minimize error propagation by correcting the earliest error.
- **Minimize Control Flow Changes:** Repairs that change the control flow (such as returning immediately or skipping a call) are prioritized after repairs that only affect the state.

A repair may eliminate one failure, only to expose another failure. In this case, ClearView performs the full process of finding correlated invariants, generating candidate repair patches, and evaluating the generated patches all over again, starting with the patched application. ClearView may therefore generate multiple patches to repair multiple invariants. This actually happened in the Red Team exercise (see Section 4.4).

If it cannot find a successful repair in the current procedure, ClearView repeats the repair evaluation process for each procedure on the call stack.

# 3. Application Communities

It is possible to apply our technique successfully whenever repeated exposures to an error or attack give ClearView the opportunity to learn how to defend against the error or attack. One appropriate deployment environment is an *application community* — a group of machines running the same application that work together to detect and eliminate failures and/or to defend themselves against attacks.

Such a monoculture is convenient for users because it provides them with a familiar software environment across all machines, thereby making their data and expertise portable across the entire computing infrastructure. It can also decrease the system administration overhead. However, it may also be convenient for attackers, who may be able to exploit a single vulnerability throughout the entire community. By configuring ClearView to protect an application community, we view the software monoculture not as a

weakness, but as an opportunity to enhance the effectiveness of the countermeasures that ClearView can deploy to neutralize attacks. An application community provides the following benefits:

- **Learning Accuracy:** The learning component can work with many different users, datasets, and usage styles, thereby increasing the accuracy of the learned invariants and the quality of the applied repair patches.
- **Amortized Learning Overhead:** ClearView can distribute the learning overhead across the community, with each member incurring overhead for only a small part of the application.
- **Faster Repair Evaluation:** The community can evaluate candidate repairs in parallel, reducing the time required to find a successful repair.
- **Protection Without Exposure:** After some members of the community are attacked and ClearView has found a successful patch, the patch is distributed throughout the community. The remaining members of the community become immune to the attack even though they have never been exposed to the attack. Furthermore, because because the patch corrects the error, it can enable applications to immediately survive *other* attacks that attempt to exploit the same vulnerability, again with no previous exposure to these attacks (see Section 4).

Our implemented ClearView application community architecture contains components on both the community machines and a centralized server that coordinates the actions of the community. An instance of the Determina Node Manager runs on each community machine. It coordinates the application of patches to applications running on that machine and provides secure communication (via SSL) with the Determina Management Console running on the centralized server. The Management Console coordinates the distribution of patches to the Node Managers and mediates the communication between the ClearView components located on the community machines and the centralized server.

## 3.1 Amortized Parallel Learning

We extended Daikon to work in parallel across the members of a community as follows. On each community machine, a local version of Daikon processes the trace data to compute invariants that are true on that machine. ClearView periodically uploads the locally inferred invariants to the centralized server, which uses the invariants to update ClearView's centralized database of invariants that are true across all executions on all members of the community. Note that ClearView uploads only the learned invariants (and not the large trace data that the local Daikon uses to infer the invariants) to the central server.

ClearView is designed to distribute the learning overhead among the members of the community as follows. Each community machine selects some of the procedures in the application to trace, then instruments only those procedures to generate trace data. The rest of the application executes without learning (and without any learning overhead). The fact that it is possible to learn over arbitrary parts of the application enables a wide range of distributed learning strategies, with each strategy designed to optimize the tradeoff between the learning overhead at each community member, the comprehensiveness of the learning coverage, and the time required to obtain an acceptable set of invariants. One particularly effective learning strategy instruments a randomly chosen small part of every running application, with new invariants continuously trickling in from all members of the community. This strategy minimizes the learning overhead while keeping the invariants up to date with information from the latest usage patterns.

It is also possible to stage the learning. The first learning phase would record the inputs and the regions (typically procedures) they exercise in the application. The second learning phase would respond to a failure by instrumenting regions close to the failure location, then replaying inputs that exercise these instrumented regions. Daikon would then process the generated trace data to produce a set of candidate correlated invariants. The drawback of this approach is that learning only in response to failures would extend the amount of time required to obtain a successful patch. The advantage is that it would reduce the learning overhead and eliminate the need for a large invariant database.

It is important to discard any invariants from executions with errors. Our currently implemented system simply excludes invariants from erroneous executions. It is also possible to apply more sophisticated strategies, for example delaying the incorporation of newly learned invariants for a period of time long enough to make any undesirable effects of the execution apparent. Only after the period has expired with no observed undesirable effects would the system use the invariants to update the centralized invariant database.

## 3.2 Application Community Management

We next describe the actions ClearView takes as it manages the community in response to a failure.

**Detection and Failure Notification:** A ClearView monitor running on one of the community machines encounters the failure. The monitor terminates the application, then uses the underlying Determina secure communication facilities to notify the central ClearView manager of the failure. The notification includes the failure location and (if available) the call stack at the time of the failure.

**Identifying Correlated Invariants:** The central ClearView manager responds to the failure notification by using the failure location and call stack to access its central invariant database and compute a set of candidate correlated invariants. For each such invariant, it generates a snippet of C code that checks the invariant, then compiles the C code to obtain a patch that checks the invariant. It presents the patches to the Determina infrastructure, which pushes the patches out to all of the members of the community. The local Determina Node Managers apply the patches to executing and newly launched instances of the application.

As the patches execute, they generate a stream of invariant check observations that are sent back to the centralized ClearView manager. As described in Section 2.4.3, each observation identifies the invariant, the failure that caused ClearView to generate the invariant checking patch, and an indication of whether the invariant was satisfied or violated.

Eventually one or more instances of the application may encounter the failure again. When the central ClearView manager receives the failure notifications, it analyzes the invariant check observations (as described in Section 2.4.3) to compute a set of correlated invariants. It then uses the Determina infrastructure to remove the invariant checking patches — the Determina Console Manager instructs the Determina Node Managers to remove the patches from any instances of the application on their machines. ClearView currently performs this step when it receives the second failure notification from a version of the application with invariant checking patches in place. It is straightforward to implement other policies.

The current ClearView configuration always runs applications with the Shadow Stack and Heap Guard monitor turned on. It is straightforward to implement other policies. For example, one could turn these features on only after encountering the first failure, then turn them back off again after ClearView finds a patch that eliminates the failure or when the community goes for a certain period of time without observing a failure.

**Generating and Evaluating Repairs:** The central ClearView manager next generates candidate repair patches for all of the correlated invariants, specifically by generating and compiling a snippet of C code that implements the invariant check and enforcement (see Section 2.5). It then uses the Determina infrastructure to apply and remove the patches to or from running or newly launched instances of the application as appropriate to implement the ClearView repair evaluation algorithm (see Section 2.6). Ideally, this repair algorithm eventually finds a successful patch which is applied across the entire community (including instances of the application which have never encountered the failure).

**Multiple Concurrent Failures:** It is possible for the community to encounter different failures at the same time (the Red Team exercise explored such a scenario). Because all ClearView patches are applied in response to a specific failure (as identified by the failure location) and all ClearView communications identify the failure ultimately responsible for the communication, ClearView is able to successfully manage the community as it responds to the events generated in response to multiple different concurrent failures.

# 4. Red Team Exercise

As part of DARPA's Application Communities program (`www.darpa.mil/IPTO/programs/ac/ac.asp`), DARPA hired Sparta, Inc. (`www.sparta.com`) to perform an independent, adversarial Red Team evaluation of ClearView. The Red Team consisted of eleven Sparta engineers. The goal of the Red Team was to discover and exploit flaws in our approach and in the ClearView implementation. The other Red Team exercise participants consisted of the Blue Team (the authors of this paper) and the White Team (a group of engineers from Mitre, Inc. led by Chris Doh of Mitre). The White Team determined the rules of engagement and refereed the exercise. The exercise was held at MIT on February 25 - 28, 2008.

During this exercise the Red Team used ten distinct exploits to attack the application protected by ClearView (Firefox 1.0.0). All of these exploits were verified to successfully exploit a security vulnerability in the unprotected version of Firefox. ClearView detected and blocked all attacks, terminating Firefox before the attacks took effect. Moreover, ClearView generated successful patches for seven of the ten attacks. All patches that ClearView's evaluation mechanism determined to be effective were, in fact, successful: they enabled the application to survive the attack and continue to execute successfully after the attack.

## 4.1 Evaluation Goals

The primary purpose of the Red Team exercise was to evaluate the effectiveness of the ClearView technology in protecting against binary code injection attacks, i.e., attacks that attempt to subvert the control flow of the application, typically by causing the application to jump to downloaded code, but more generally by causing the application to take any unauthorized control flow transfer. This particular class of attacks was chosen because they are common in practice and can have particularly serious consequences if successful. The Red Team evaluation had several specific goals:

- **Surviving Attacks:** Determine if ClearView can respond to attacks by finding patches that enable the application to survive the attack and continue to execute successfully.
- **Repair Evaluation:** Determine if ClearView can generate a patch that impairs the application in some way, either by causing it to behave incorrectly on legitimate inputs or by creating a new exploitable error.
- **False Positives:** Determine if legitimate inputs can trigger the ClearView patch generation mechanism.
- **Infrastructure Attacks:** Determine if attackers can subvert the ClearView patch generation and distribution mechanism to send out malicious patches. This paper omits the detailed results of this qualitative evaluation, but in summary the standard security measures already in place in the Determina commercial product (encryption, authentication, etc.) were judged to provide an acceptable level of protection against this class of attacks.

## 4.2 Rules of Engagement

The rules of engagement determined the scope of the Red Team exercise — what kinds of Red Team attacks were in bounds, how to judge if an attack succeeded or failed, the access that the Red Team was given to Blue Team information, etc. Together, the Red, Blue, and White Teams agreed on an application (the unmodified, stripped x86 binary of Firefox 1.0.0) for the Blue Team to protect. With this application, the attack vector was web pages — the Red Team launched all attacks by navigating Firefox to one or more attack HTML, XUL, or GIF files. Firefox 1.0.0 has several properties that made it appropriate for this exercise:

- **Mature Code Base:** The Firefox code base was relatively mature and tested, which made it a reasonable proxy for other mature applications that ClearView is designed to protect.
- **Vulnerabilities:** This version of Firefox contained enough vulnerabilities to support a thorough evaluation without the need for the Red Team to find an infeasibly large number of new vulnerabilities.
- **Source Code Availability:** Source code was available for this application. Although ClearView does not require (or even use) any source information, the availability of source code made it much easier to understand the behavior of the application and interpret the phenomena observed during the Red Team exercise.
- **Automation:** Firefox supported the automated loading of web pages, which facilitated automated learning and testing both in preparation for and during the Red Team exercise.
- **High Availability Requirements:** In the future, much of the Internet content that users will need or want to access may contain exploits (see Section 6.2). Terminating Firefox when it encounters an attack (or filtering out content with exploits) may therefore substantially impair its utility.

Given the envisioned scope of the Red Team exercise and the available resources, it was not feasible to add more applications to the Red Team exercise.

### *4.2.1 Attack Scope*

The Red Team attacks fall into three categories: control flow attacks, induced autoimmune attacks, and false positive attacks. Control flow attacks attempt to subvert the flow of control within the application. Such an attack was judged to succeed if it prevented the application from continuing to successfully process additional inputs, either by successfully redirecting the flow of control to malicious code or by causing the application to crash.

A false positive attack succeeds when it causes ClearView to apply a patch in response to loading a legitimate web page.

Induced autoimmune attacks attempt to turn the ClearView patch mechanism against the application. Such an attack succeeds if it ClearView's patch affects the behavior of the application on legitimate inputs (as opposed to attack inputs). An autoimmune attack was judged to succeed if the patched version of Firefox, when made to navigate to a sequence of legitimate web pages, did not behave the same as the unpatched version (bit-identical displays, same user functionality).

Within these constraints, the Red Team was given completely free rein in generating attacks. Known attacks, variants on known attacks, completely new attacks, and attacks that involved different web pages loaded in sequence were all within scope. There were no restrictions whatsoever placed on the information that the Red Team was allowed to use when generating attacks.

### *4.2.2 Red Team Exercise Preparation*

Prior to the Red Team exercise, the Blue Team generated an invariant database by running Firefox on a collection of twelve web pages that exercise functionality related to known Firefox vulnerabilities. Learning was confined to selected regions of the application related to these vulnerabilities. The web pages and invariant database were both made available to the Red Team before the Red Team exercise.

Several months before the Red Team exercise, the Blue Team provided the Red Team with all of the Blue Team's source code, tests, and documentation, including design documents, presentations to sponsors, and the Blue Team's own analyses of weaknesses in ClearView. During the period of time leading up to the Red Team exercise, the Blue Team periodically provided the Red Team with source code, test, and documentation updates. At the time of the Red Team exercise, the Red Team had complete access to all of the source code, tests, and documentation for the running Blue Team system.

Prior to the Red Team exercise, the Red Team selected 57 evaluation web pages. These legitimate web pages exercise a range of Firefox functionality and were used during the Red Team exercise for repair evaluation (specifically, to determine if the patched version of Firefox displayed the evaluation pages correctly) and false positive evaluation (specifically, to determine if any of the evaluation pages triggered the ClearView patch generation mechanism). The Blue Team was not provided with these web pages prior to the Red Team exercise.

For the Red Team exercise, the Blue Team provisioned a small community of machines with Firefox deployed on all machines. The Blue Team configured ClearView with Memory Firewall, Heap Guard, and the Shadow Stack enabled from the start on all Firefox executions.[2] The Red Team attacked this community during the Red Team exercise.

## 4.3 Attack Evaluation

The first phase of the Red Team exercise evaluated ClearView's ability to protect Firefox against Red Team attacks. The Red Team selected ten defects in Firefox, then created one or more exploits for each defect. The targeted defects cause exploitable errors such as unchecked JavaScript types, out of bounds array accesses, heap and stack buffer overflows, and JavaScript garbage collection problems. All of the exploits were verified to work — they successfully exploited a vulnerability in Firefox. The Red Team used the exploits to perform the following attacks.

### *4.3.1 Single Variant Attacks*

For each defect, the Red Team chose an exploit, then repeatedly presented the exploit to an instance of Firefox running in the community. The Red Team presented each attack only after ClearView had performed all actions taken in response to the previous attack. For all ten exploits, ClearView monitors detected and blocked the corresponding attacks. For seven of the exploits, ClearView generated patches that enabled Firefox to continue to execute through the attacks to correctly display the subsequently loaded evaluation pages. The Red Team observed no differences between the patched and unpatched versions of Firefox. Subsequent investigation after the Red Team exercise (see Section 4.3.2) indicated that small configuration changes enabled ClearView to successfully generate patches for two of the remaining three exploits.

[2]With one exception. Specifically, the presence of Heap Guard disabled the exploit for the defect with Bugzilla number 296134. To enable the meaningful inclusion of this exploit in the Red Team exercise, the Blue Team turned off Heap Guard when the Red Team deployed this exploit.

| Bugzilla Number | Presentations | Error Type |
|---|---|---|
| 269095 | 6 | Memory Management |
| *285595 | 4 | Heap Buffer Overflow |
| 290162 | 4 | Unchecked JavaScript Type |
| 295854 | 5 | Unchecked JavaScript Type |
| 296134 | 4 | Stack Overflow |
| 311710 | 12 | Out of Bounds Array Access |
| 312278 | 4 | Memory Management |
| 320182 | 6 | Memory Management |
| *325403 | 4 | Heap Buffer Overflow |

**Table 1: Number of times each exploit was presented before ClearView created and applied a patch that protected against the exploit. A * identifies the two exploits for which ClearView did not successfully generate a patch during the Red Team exercise, but did successfully generate a patch in subsequent experiments after reconfiguration.**

Table 1 presents the number of exploit presentations required for ClearView to find and apply the patch that enabled Firefox to execute successfully through the attack. In general, the minimum number of exploit presentations is four. The first presentation makes ClearView aware of the exploit; ClearView responds by computing a set of candidate correlated invariants and applying patches that check candidate invariants (see Sections 2.4.1 and 2.4.2). During the next two presentations ClearView records invariant satisfaction and violation information to compute the set of correlated invariants (see Section 2.4.3). After these presentations ClearView removes the invariant checking patches and generates and applies patches that enforce correlated invariants (see Sections 2.5 and 2.6). If the first invariant enforcement patch is successful, ClearView has corrected the error in four presentations.

As Table 1 indicates, the first invariant enforcement patch successfully corrected the errors from exploits 290162, 296134, 312278, 285595, and 325403. For exploit 295854 the first patch did not correct the error, but the second patch did. For exploits 269095 and 320182 the third patch was the first successful patch.

Exploit 311710 is an outlier in that it involves three separate defects, each of which is exploited by the same attack. ClearView corrects the error from the first defect after four presentations, at which point the attack exploits the second defect. It takes ClearView another four presentations to correct the error from this second defect, at which point the attack exploits the third defect. It takes ClearView another four presentations to correct the error from this final defect, for a total of twelve presentations to obtain a set of patches that enables Firefox to successfully survive the attack.

**Stack Overflow:** Exploit 296134 causes Firefox to erroneously compute a negative value for the length of a string. This negative length is then passed to `memcpy`, which treats it as a very large unsigned integer. The resulting copy writes downloaded data over exception handlers on the stack. When the copy proceeds past the end of the stack, the invoked overwritten exception handler executes downloaded code.

During learning ClearView learned a lower-bound invariant that requires the computed string length to be at least one. ClearView generated a patch that enforced this invariant by setting the length

to one. This patch corrected the out of bounds writes to the stack and enabled Firefox to survive the attack.

**Unchecked JavaScript Type Exploits:** Both exploits 290162 and 295854 download JavaScript code that creates an object, then fills the object with malicious code and data. A JavaScript system routine fails to check the type of the object and (eventually via a sequence of operations) invokes downloaded code via a C++ virtual function call on the corrupted object.

During learning ClearView learned a one-of invariant at the virtual function call site for both errors. These invariants state that the call site may invoke only one of the functions invoked at that site during learning. The first patch that ClearView applied during repair evaluation enforced the invariant by invoking a specific previously invoked function instead of jumping to malicious code. This patch successfully corrected the error from exploit 290162, but failed to correct the error from exploit 295854. ClearView's second applied patch, which enforced the invariant by skipping the call, successfully corrected the error from exploit 295854.

**Memory Management Exploits:** Exploit 312278 enables downloaded JavaScript code to obtain a pointer to an object that is erroneously garbage collected, then reallocated to hold a native Firefox C++ object. The downloaded JavaScript code then overwrites the C++ object's virtual function table pointer with a pointer to memory containing pointers to malicious downloaded code. During learning ClearView learned a one-of invariant at the virtual function call site that invokes the malicious code. This invariant states that the call site may invoke only one of the functions invoked at that site during learning. The first patch that ClearView applied during repair evaluation successfully corrected the error by invoking a specific previously invoked function.

Exploits 269095 and 320182 involve memory that is not reinitialized after it is reallocated. Under certain circumstances it is possible to manipulate Firefox into treating this uninitialized memory as a C++ object, then invoking a virtual function call on this uninitialized object. Downloaded JavaScript code can exploit this error to fill the memory with appropriately formatted malicious code and pointers before it is reallocated. In this case the virtual function call invokes the malicious code. During learning ClearView learned a one-of invariant at the virtual function call site that invokes the malicious code. One of the patches that ClearView applied during repair evaluation enforced the invariant by returning from the function that contains the call site before the call site is invoked. This patch enabled Firefox to survive the attack.

Before trying this patch, ClearView tried patches that invoke one of the previously observed functions and a patch that skips the call but executes the remaining part of the function following the call. Neither of these patches enabled Firefox to survive the attack.

**Out of Bounds Array Access Exploits:** Exploit 311710 causes Firefox to compute a negative array index, then use the index to attempt to retrieve a C++ object from the array. Downloaded JavaScript code previously caused the retrieved memory to contain pointers to downloaded code. When Firefox performs a virtual function call on the retrieved object, it invokes the downloaded code.

During learning ClearView learned a lower-bound invariant that requires the array index to be non-negative. During repair evaluation ClearView applied a patch that enforces this invariant by setting the array index to zero. This patch caused Firefox to retrieve a valid C++ object, the resulting virtual function call invoked valid code, and Firefox survived the attack.

The same defect that caused this error was present in three similar procedures (apparently created via copy and paste) that executed during the attack. A similar patch corrected all of the errors from these defects.

### *4.3.2 Remaining Exploits*

During the Red Team exercise, ClearView did not generate a successful patch for three of the Red Team's exploits. Exploit 285595 targets code for an unused Netscape GIF extension. Because this code does not check the sign of a value extracted from the GIF file, it is vulnerable to a remotely exploitable heap overflow attack. During the Red Team exercise, ClearView's correlated invariant identification component was configured to consider invariants from only the lowest procedure on the stack with invariants. The relevant invariant appeared one procedure above this procedure. ClearView therefore did not produce a patch that corrected the error. We subsequently verified that changing the configuration to include additional procedures on the stack enabled ClearView to generate a successful patch that corrects this error. The relevant invariant is a lower-bound invariant involving a buffer index. The repair changes the index from a negative value to zero, thereby bringing out of bounds writes back into the buffer. The exploit itself is embedded within an image file. The repair neutralizes the attack and enables Firefox to display the image correctly.

Exploit 325403 involves a downloaded HTML file that can specify a buffer growth size for data that does not fit in an allocated buffer that holds two-byte Unicode characters. By specifying a buffer growth size very close to the largest number that fits in an integer, an attacker can cause the calculation of the new buffer size to overflow, causing Firefox to allocate a buffer that is too small. An ensuing `memcpy` then writes beyond the end of the allocated buffer. The Blue Team's learning suite for the Red Team exercise did not provide sufficient coverage for Daikon to learn the relevant invariant. We subsequently verified that, using an expanded learning suite, Daikon would have learned an invariant that would have enabled ClearView to generate a successful patch. The relevant invariant is a less-than invariant relating the buffer size to the size of the memory to copy into the buffer. The repair sets the copy size to the buffer size, eliminating the out of bounds writes.

Exploit 307259 causes Firefox to compute an incorrect size for a buffer holding a hostname that contains soft hyphens. When Firefox attempts to copy a number of items into this buffer, the copies write beyond the end of the buffer. ClearView did not generate a successful patch because Daikon's invariants are not rich enough to capture the error. The appropriate invariant would generalize Daikon's less-than invariant (which relates two quantities) to relate a sum of buffer lengths to another buffer length. Learning richer invariants would be possible, but would increase the cost of the learning component.

### *4.3.3 Comparison With Manual Fixes*

Manual fixes are available for the defects that the exploits in the Red Team exercise exploited. For exploit 269095, the manual fix tags deallocated objects as invalid. Subsequent object uses check this tag. If the tag is invalid, the use returns an error. The fix also iterates over invalid objects to reinitialize relevant data. For exploit 285595, the manual fix removes the code containing the defect. This code implemented a Netscape GIF extension; the fix removes support for this extension from Firefox. For exploits 290162 and 295854, the manual fix checks the type of the JavaScript object. If the check fails, the enclosing method (which otherwise invokes a method on the object) simply returns null. For exploit 296134, the manual fix adds a check for negative string length. If the check fails, the enclosing method logs an error, returns, and does not perform the copy. The fix also includes a check in the calling method that truncates the string length to the allocated buffer size. The manual fix for 311710 corrects a conditional that caused the application to compute the negative array index.

For exploit 312278, the manual fix informs the garbage collector that it holds a reference to the relevant object. Once the garbage collector is aware of this reference, it does not collect the object and the memory holding the object is unavailable to the JavaScript code in the exploit. For exploit 320182, the manual fix sets a flag that identifies reallocated objects. Subsequent code checks the flag to identify and properly initialize any such reallocated objects. For exploit 325403, the manual fix checks that the target array is large enough to hold the data in the source array. If the check fails, the fix allocates a larger target array and retries the copy.

Some of these manual fixes perform a consistency check close to the error, then skip the remaining part of the operation if the check fails. All ClearView patches similarly perform a consistency check (the invariant satisfaction check) close to the error. Two of the ClearView repairs (skip call and return from enclosing procedure) have a similar effect of explicitly skipping part or all of the remaining part of the operation. Other repairs (adjusting values to enforce lower-bound and less-than relationships) often have the effect of enabling the application to execute the remaining part of the operation safely with the effect of any remaining errors localized to that operation. In general, the ClearView repairs tend to execute more of the normal-case code following the error, while the manual fixes tend to simply abort the current operation. We attribute this more drastic approach to the maintainer attempting to simplify the reasoning required to confirm that the fix has eliminated the error.

It would be possible to enhance ClearView to produce checks and repairs that more closely correspond to these manual fixes. For example, it would be possible to enhance ClearView to automatically infer object types at dynamically dispatched method invocations and to return error codes from enclosing procedures when invariant checks fail. It would also be possible to develop repairs that skip larger parts of the subsequent computation.

Some of the manual fixes inform the garbage collector of existing references and reinitialize recycled memory. These fixes affect code far from the failure location. ClearView would therefore need to apply a more sophisticated correlated invariant identification strategy (and potentially more sophisticated invariants as well) to produce repairs with similar effects.

#### *4.3.4 Multiple Variant Attacks*

For three defects, the Red Team generated multiple variants of the exploit that targeted the defect. For each defect the Red Team interleaved different variants during the attack. ClearView generated the same patch after the same number of attacks as for the corresponding single variant attack. This patch successfully protected Firefox against all variants of the attack.

#### *4.3.5 Simultaneous Multiple Exploit Attacks*

The Red Team launched several attacks that interleaved exploits that targeted different defects. The goal was to determine if targeting different defects with different exploits would impair ClearView's ability to generate successful patches. In each case ClearView was able to determine the targeted error for each attack, keep the learning data separate for the different errors, and generate a set of patches that together successfully protected Firefox against all of the exploits in the attack. And ClearView was able to generate these patches after the same cumulative number of attacks as for the corresponding sequence of single variant attacks.

#### *4.3.6 Repair Evaluation*

For all of the previous attack scenarios the Red Team evaluated the quality of the repair by determining whether the patched version of Firefox displayed the evaluation web pages correctly (which it always did).

The final repair evaluation started with applying all of the successful patches generated during the previous attacks to Firefox. The Red Team then used this patched version of Firefox to display all of the evaluation pages. In each case the displays were identical to the displays produced by the unpatched version, with no other behavioral change that the Red Team could detect. This result indicates that the Red Team was unable to launch a successful induced autoimmune attack.

In general, the ClearView repair evaluation mechanism is designed to find and discard patches that have a negative effect on the application. And in fact, ClearView did generate patches with negative effects (such as causing the application to crash) during the Red Team exercise. But the repair evaluation mechanism detected and discarded these patches, mitigating the effect on the application and paving the way for the application of successful patches.

#### *4.3.7 False Positive Evaluation*

The Red Team's false positive evaluation used ClearView to display the evaluation web pages. The goal was to make ClearView generate an unnecessary patch. During this evaluation ClearView generated no patches at all, indicating that the Red Team was unable to cause ClearView to produce a false positive.

### 4.4 Performance

The Red Team exercise used a Dell 2950 rack-mount machine with 16 GB of RAM and two 2.3 GHz Intel Xeon processors, each with four processor cores. We ran Firefox inside VMware virtual machines under ESX servers. The operating system was Windows XP Service Pack 2. Because the Red Team's exploit 296134 has no effect in this environment, we ran this exploit on a 1.8 GHz AMD Opteron machine with four processor cores and 8 Gbytes of RAM running Windows XP Service Pack 2. In this environment the exploit does trigger the error. Because the exploit is running on a slower computing platform, the execution times for the various activities are, in general, proportionally longer than the corresponding times for other exploits.

#### *4.4.1 Learning Overhead*

The time required to load the twelve learning web pages without learning enabled was 5.2 seconds. The time required to load these same web pages with learning enabled was 1600 seconds (over a factor of 300 slower). The Daikon x86 front end, which records and dumps the values of accessed memory locations and registers, is responsible for the vast majority of the overhead. As described in Section 3.1, it is possible to distribute the learning in parallel across the application community.

#### *4.4.2 Baseline Overheads*

Table 2 presents the times required for Firefox to load the 57 evaluation pages from a local disk with the network interface disabled when running under various ClearView configurations. We ran the experiments on an Intel Core 2 Duo E6700 (2.66 GHz, 4 MB L2 Cache, 3.25 GB RAM) running Windows XP Service Pack 3. The Determina Managed Program Execution Environment (with Memory Firewall enabled) imposes a 47% overhead over running Firefox as a standalone application. This overhead is somewhat larger than typically observed [4]. We attribute this additional overhead to Firefox's use of object-oriented constructs such as dynamic method dispatch. With standard hardware jump optimization techniques, the underlying DynamoRIO code cache implementation of the resulting indirect jump instructions is relatively less efficient than implementations of other instruction patterns [4].

| 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 |

**Table 2: Page load times and overheads for different ClearView configurations running Firefox.**

### *4.4.3 Patch Generation Time*

On average ClearView took 4.9 minutes from the time of the first exposure to a new exploit to the time when it obtained a successful patch for that exploit. This is not the time required to stop a propagating attack — Memory Firewall terminates the application before the attack can take effect, so there is no propagation. These times instead reflect how long users must wait before they have a patched version of the application that provides continuous, uninterrupted service even while under attack.

The 4.9 minutes includes an average of 5.4 executions: to detect the failure and select a set of candidate correlated invariants, to collect invariant checking results to identify correlated invariants, and to evaluate candidate repairs. These averages include one outlier, for which ClearView took 13 minutes to sequentially correct three distinct errors in the application, all of which were exploited by the same attack — after ClearView repaired one error, the same exploit triggered an error from a different defect which ClearView then detected and repaired, and so on.

### *4.4.4 Patch Creation Time Breakdowns*

When considering how much time ClearView takes to create a successful repair, one key comparison to keep in mind is the 28 days (on average) that it takes for developers to create and distribute a patch for a security exploit [41]. This section breaks down ClearView's 4.9 minutes (on average) time to do the same.

Table 3 presents the different components of the time ClearView requires to generate a successful repair. All times are in seconds. With the exception of exploit 311710, there is one row for each exploit. The first column of each row presents the Bugzilla number of the exploit. Exploit 311710 has three rows (labeled 311710a, 311710b, and 311710c). As described above, ClearView corrected three distinct errors before enabling Firefox to finally survive the attack. We place each error in a separate row.

**Shadow Stack, Heap Guard Runs:** The second column of each row (Shadow Stack, Heap Guard Runs) in Table 3 presents the time required to replay the exploit to detection with the Shadow Stack and Heap Guard turned on. The vast majority of this time (20–30 seconds) is spent warming up the Determina Managed Program Execution Environment code cache when we restart Firefox to process the exploit. For all exploits except 311710, this time is roughly 20 to 30 seconds. Because exploit 311710 exercises more Firefox functionality than the other exploits, the times are higher for this exploit. Our presentation of the numbers in this column makes it possible to compute successful patch generation times for a deployment environment in which only Memory Firewall is enabled during production use, with Heap Guard and the Shadow Stack enabled only after the detection of the first attack.

We note that it is possible for ClearView to successfully correct errors with only Memory Firewall enabled. And in fact, the use of Heap Guard did not improve ClearView's performance in the Red Team exercise — Memory Firewall and Shadow Stack are all that is required for ClearView to generate successful patches for the seven exploits that it successfully patched during the Red Team exercise. Heap Guard is required for the remaining two exploits for which ClearView was able to subsequently generate successful patches after configuration changes.

**Building and Installing Invariant Checks:** The Building Invariant Checks column presents the time required to build all of the invariant check patches. This time includes compiling the automatically generated C source code for the patches and loading the patches into a DLL for presentation to the Determina patch management system. Each entry has the form $t[x,y,z]$, where $t$ is the time required to build the invariant checks, $x$ is the number of checked one-of invariants, $y$ is the number of checked lower-bound invariants, and $z$ is the number of checked less-than invariants. Exploit 296134 is an outlier, in part because ClearView compiled many more invariant check patches than for the other exploits and in part because the compiles took place on a slower computing platform. The Installing Invariant Checks column presents the time required for the Determina patch management system to transmit and apply the patches to the application running on the client machine.

**Invariant Check Runs:** The Invariant Check Runs column presents the time required to replay the exploit to detection twice with the invariant check patches in place. Each entry has the form $t(x/y)$, where $t$ is the time required to replay the exploit to detection twice, $x$ is the number of times a checked invariant was violated during these runs, and $y$ is the total number of invariant checks executed during these runs. The time $t$ includes the time required to communicate the necessary invariant check, shadow stack, and attack location information to the ClearView Manager. As before, much of the time was spent warming up the Determina Managed Program Execution Environment code cache. For exploit 296134 ClearView also spent a substantial amount of time communicating invariant check results to the ClearView Manager using the Windows event queue mechanism.[3]

**Building and Installing Repair Patches:** The Building Repair Patches column presents the time required to build all of the repair patches for the correlated invariants to evaluate. Each entry has the form $t[x,y,z]$, where $t$ is the time required to build the repair patches, $x$ is the number of correlated one-of invariants, $y$ is the number of correlated lower-bound invariants, and $z$ is the number of correlated less-than invariants.[4] All of the correlated one-of invariants involve function pointers. As described above in Section 2.5.1, such invariants have three potential repairs. ClearView compiles a patch for each such repair. The other kinds of invariants each have a single repair with a single repair patch. The Installing Repair Patches column presents the time required to communicate these patches to the client machine. The client machine applied specific repair patches one at a time in response to directives from the central ClearView Manager.

**Unsuccessful Repair Runs:** The Unsuccessful Repair Runs column presents the time (if any) required to execute Firefox to completion for any unsuccessful repair patches. Each entry has the form $t(x)$, where $t$ is the time required to execute Firefox to completion and $x$ is the number of unsuccessful runs (if any). We attribute the small number of unsuccessful runs to the effectiveness of the correlated invariant selection policy in targeting invariants whose repairs are likely to correct the error and to the effectiveness of the candidate repair ordering rule in selecting an effective repair to evaluate first.

[3]For exploit 296134 we are missing the number of times that the invariant checks were executed and the number of times that the corresponding invariants were violated.

[4]For exploit 296134 we are missing the number of correlated lower-bound and less-than invariants.

| 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 |

**Table 3: ClearView attack processing times, in seconds. Because all timing events were measured on the central ClearView Manager, the times include communication times between the protected client and the manager. Attacks for exploit 296134 were run on a slower computer (see Section 4.4.4). A * identifies the two exploits for which ClearView did not successfully generate a patch during the Red Team exercise, but did successfully generate a patch in subsequent experiments after reconfiguration. A ! identifies the exploit for which ClearView did not successfully generate a patch in either the Red Team exercise or in subsequent experiments.**

**Successful Repair Run:** The Successful Repair Run column shows the time required to execute Firefox with the successful repair patch applied. Because this patch corrects the error and eliminates the attack detection, the patch was judged to succeed ten seconds after it executed with no subsequent attack detection. The presented time includes this ten seconds. At this point ClearView generated and identified a patch that corrected the error and enabled continued successful execution. The final column is the sum of the times in the other columns. It presents the total time required to automatically obtain a successful patch for the corresponding attack.

### *4.4.5 Overhead Reduction Techniques*

There are three primary sources of inefficiency in the current ClearView attack response system: warming up the Determina Managed Program Execution Environment code cache, using Windows event queues as the communication mechanism between community members and the centralized server, and compiling the invariant check and repair patches. It is possible to eliminate the cache warm up time by saving the cache state from a previous run, then restoring this state upon startup. It is possible to dramatically reduce the communication time by using a more efficient communication mechanism. It is possible to eliminate the compilation time by generating binary code directly instead of generating, then compiling C code. Together, we estimate that these optimizations would enable ClearView to produce successful patches in tens of seconds rather than minutes.

## 4.5 Other Applications

In general, we believe the Firefox results to be broadly representative of the results ClearView would deliver for other server applications. Many of the ClearView patches have a similar effect as other techniques such as failure-oblivious computing [29, 27, 30] and transactional function termination [36, 35]. Specifically, enforcing lower-bound and less-than invariants may coerce otherwise out of bounds references back within the bounds of the accessed block of memory. Like failure-oblivious computing, this coercion localizes the effects of out of bounds write errors and prevents otherwise fatal global corruption of the application state. Like transactional function termination, enforcing one-of invariants involving function pointers by skipping calls or returning from the calling procedure can eliminate the execution of the code that contains an otherwise exploitable defect. Both failure-oblivious computing and transactional function termination have been shown to be effective in enabling a range of servers to successfully survive a range of errors and attacks.

We note that because ClearView is based on inferred invariants (which may capture aspects of the application's semantics) it may, in principle, be able to generate more targeted and therefore more effective repairs. ClearView also incorporates a broader range of repair strategies and evaluates the resulting multiple candidate repairs to discard ineffective or damaging repairs, which may enhance its ability to find successful patches.

We have less experience applying survival techniques to applications other than servers. In general, we anticipate that survival strategies will be more effective for the broad range of applications (such as applications that process sensory data and information retrieval applications) that can tolerate some variation in the result that the computation produces. We anticipate that survival techniques may be less appropriate for applications (such as compiler transformations) with precise, logically defined correctness requirements, long dependence chains that run through the entire computation, and less demanding availability requirements.

# 5. Limitations

The goal of 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.

Of course, there are errors that are completely outside the scope of ClearView, i.e., errors for which there is no plausible learned invariant whose enforcement would enable the application to survive. But even if the error is within the scope of the overall ClearView approach, ClearView may be unable to find a repair that enables the application to survive the error:

- **Learning:** Daikon comes preconfigured to learn a specific set of invariants. This set may not include an invariant that enables ClearView to generate a patch for a given error. And even if the set does include such an invariant, the learning phase may not provide enough coverage of the application to enable Daikon to learn this invariant.
- **Monitoring:** ClearView currently uses Memory Firewall to detect control flow transfer errors and Heap Guard to detect out of bounds write errors. Additional detectors would be required to detect other kinds of errors.

- **Candidate Invariant Selection:** Every repair enforces an invariant that ClearView selected as a candidate correlated invariant. Even if ClearView inferred an invariant with a repair that would correct the error, the failure may occur sufficiently far from the error for ClearView to not include the invariant in the set of candidate correlated invariants.
- **Repair:** The repair mechanism comes with a specific set of invariant enforcement mechanisms; each such mechanism corresponds to a specific repair strategy. It is possible for none of these repair strategies to produce a successful repair.

It is also possible for ClearView to impair the functionality of the application or even to create new vulnerabilities:

- **Functionality Impairment:** It is possible for a ClearView repair patch to impair the functionality of the application. If the patch is applied in response to a legitimate attack, the functionality impairment may be a reasonable price to pay for eliminating the vulnerability. The fact that ClearView applies patches only in response to a detected failure (in our current implementation, an illegal control flow transfer or out of bounds write) and the fact that ClearView enforces the invariant only if it is correlated with the failure minimize the likelihood that ClearView will apply a patch in the absence of an error or attack. These facts also minimize the likelihood that an applied ClearView patch will interfere with the processing of a legitimate input. And the ClearView patch evaluation mechanism enables ClearView to recognize and discard patches that do not eliminate the failure or cause the application to crash.
  Note that ClearView did, in fact, generate several patches with negative effects during the Red Team exercise. The ClearView patch evaluation mechanism detected these negative effects and discarded the patches.
- **Patch Subversion:** It is theoretically possible for an adversary to subvert the ClearView patch mechanism to install its own malicious patches. We note that ClearView builds on the commercially deployed Determina patch distribution mechanism, which uses standard authentication and encryption mechanisms to ensure patch integrity.
- **Malicious Nodes:** It is theoretically possible for a malicious node or nodes to provide ClearView with erroneous information that may cause it to generate an inappropriate patch. It is possible to mitigate this possibility by reproducing the error and evaluating the generated patches on trusted nodes before distributing the patches throughout the community.

# 6. Related Work

We discuss additional related work in attack detection mechanisms, automatic filter generation, checkpoint and replay techniques, and error tolerance and correction.

## 6.1 Attack Detection Mechanisms

ClearView uses two attack detection techniques: program shepherding to detect and block malicious control flow transfers, and heap overflow checks to detect and block out of bounds writes to the heap. In general, however, ClearView can work with any attack detection technique that provides an attack location. StackGuard [9] and StackShield [39], for example, use a modified compiler to generate code to detect attacks that overwrite the return address on the stack. StackShield also performs range checks to detect overwritten function pointers. Researchers have also built compilers that insert bounds checks to detect memory addressing errors in C programs [2, 43, 6, 18, 31, 19, 20]. Drawbacks of these techniques include the need to recompile the program, the overhead of the dynamic bounds checks, and, in some cases, the need to change the program itself [6, 20]. Dynamic taint analysis finds appearances of potentially malicious data in sensitive locations such as function pointers or return addresses [40, 10, 24]. It would be possible to make ClearView work with all of these detectors, although the high overhead and potential need for recompilation or even source code changes goes against ClearView's philosophy of operating on stripped Windows binaries and minimizing the overhead during normal execution.

The large overhead of many proposed protection techniques has inspired attempts to reduce the performance impact on applications in production use. One commonly proposed technique is to run heavily instrumented versions of potentially vulnerable applications on honeypots [38, 33, 1], leaving the production versions unprotected against new attacks. When the honeypot is attacked, the instrumentation can detect the attack and develop a response that protects the production versions of the application. This technique can be applied to virtually any attack detection and response mechanism that is too expensive to deploy directly on applications running in production. It may also be possible to deploy expensive attack detection mechanisms in a piecemeal fashion across a community of machines, with each application instrumenting only a small portion of its execution [23].

In contrast to systems which apply their attack analyses across broad ranges of the application, ClearView uses the attack location to dramatically narrow down the region of the application that it instruments during its attack analysis and response generation activities. This makes it possible to deploy sophisticated but expensive analyses within this focused region of the application while still keeping the total overhead small. A potential drawback of this focus is that ClearView may miss the invariant required to correct the underlying error in the application logic.

## 6.2 Automatic Filter Generation

A standard way to protect applications against attacks is to develop filters that detect and discard exploits before they reach a vulnerable application. Vigilante uses honeypots to detect attacks and dynamically generate filters that check for exploits that follow the same control-flow path as the attack to exploit the same vulnerability [8]. Bouncer uses symbolic techniques to generalize Vigilante's approach to filter out more exploits [7]. ShieldGen uses Vigilante's attack detection techniques to obtain exploits [11]. It generates variants of each exploit and tests the variants to see if they also exercise the vulnerability. It then produces a general filter that discards all such variants.

Sweeper uses address randomization for efficient attack detection [42]. This technique is efficient enough to be deployed on production versions of applications, but provides only probabilistic protection and therefore leaves applications still vulnerable to exploitation [32]. Sweeper uses attack replay in combination with more expensive attack analysis techniques such as memory access checks, dynamic taint analysis, and dynamic backward slicing. It uses the information to generate filters that discard exploits before they reach vulnerable applications. Sweeper also builds vulnerability-specific execution filters, which instrument selected instructions involved in the attack to detect the attack. The attack response is to use rollback plus replay to recover from the attack.

Discarding inputs that may contain exploits can deny users access to content that they need or want to access. It is entirely possible, for example, for useful information sources such as legitimate web pages, images, or presentations to become surreptitiously infiltrated with exploits. Attackers may also create attractive content

specifically for the purpose of enticing users to access (via vulnerable applications) otherwise desirable or useful content containing exploits. Consider that much of the content currently available on the Internet is created for the purpose of generating advertising revenue. An alternate (and presumably illegal) business model would simply substitute attacks for advertisements. In all of these cases it is desirable for applications to process inputs containing exploits without enabling the attacks to succeed.

By enabling applications to execute successfully through otherwise exploitable errors, ClearView can enable users to access useful or desirable information even if the input containing the information also contains an exploit. The ClearView repair for one of the heap overflow errors in the Red Team exercise, for example, makes it possible for users to view useful or attractive image files that also contain exploits (or that contain innocent data that happens to exercise the vulnerability).

## 6.3 Checkpoint and Replay

A traditional and widely used error recovery mechanism is to reboot the system, with operations replayed as necessary to bring the system back up to date [17]. It may also be worthwhile to recursively restart larger and larger subsystems until the system successfully recovers [5]. Checkpointing [22] can improve the performance of the basic reboot process and help minimize the amount of lost state in the absence of replay. Checkpointing also makes it possible to discard the effects of attacks and errors to restore the system to a previously saved clean operational state. This approach, in some cases combined with replay of previously processed requests or operations that do not contain detected exploits, has been proposed as an attack response mechanism [37, 26, 34, 35, 42].

In comparison with ClearView's approach of continuing to execute through attacks, checkpoint plus replay has several drawbacks. These include service interruptions as the system recovers from an attack (these service interruptions can occur repeatedly unless the system is otherwise protected against repeated attacks), the potential for replay to fail because of problematic interactions with external processes or machines that are outside the scope of the checkpoint and replay mechanism, lost state if the system chooses to forgo replay, and complications associated with applying checkpoint and replay to multithreaded or multiprocess applications.

## 6.4 Error Tolerance and Correction

Transactional function termination [36, 33] (which restores the state at the time of the function call, then returns from functions that perform out of bounds accesses), failure-oblivious computing [29] (which discards out of bounds writes and manufactures values for out of bounds reads), boundless memory blocks [28] (which store out of bounds writes in a hash table for subsequent corresponding reads to access), and DieHard [3] (which overprovisions the heap so that out of bounds accesses are likely to fall into otherwise unused memory) are all designed to allow applications to tolerate errors such as out of bounds memory accesses. These techniques require no learning phase and no repeated executions for correlated invariant selection and evaluation. ClearView differs in applying checks and repairs only to carefully targeted parts of the application and only in response to attacks, and in performing an ongoing evaluation of each applied repair. ClearView may also be able to provide more informative error reports, since it can identify specific invariants whose violation is correlated with errors.

In a previous project [12], we developed a system that automatically inferred data structure consistency constraints and created repairs [13] to enforce them. This system was also evaluated by a (different) hostile Red Team. In contrast to ClearView, this previous system required source code, performed learning only once without subsequent refinements, applied only repairs that it statically verified to always terminate in a repaired state, and did not observe subsequent executions to evaluate the quality of the repairs.

ASSURE [35] generalizes transactional function termination to enable the system to transactionally terminate any one of the functions on the call stack at the time of the error (and not just the function containing the error). Attack replay on a triage machine enables the system to evaluate which function to terminate to provide the most successful recovery. The applied patch takes a checkpoint at the start of the function. It responds to errors by restoring the checkpoint, then returning an effective error code to terminate the function and continue execution at the caller.

Exterminator [25] uses address space randomization to detect out of bounds writes into the heap and accesses via dangling references. It then corrects the errors by (as appropriate) increasing the size of allocated memory blocks to accommodate the observed out of bounds writes or delaying memory block deallocation until the accesses via the corresponding dangling references have completed. Exterminator can combine patches from multiple users to give members of a community of users immunity to out of bounds write and dangling reference errors without prior exposure.

Researchers have used genetic programming techniques to generate and search a space of abstract syntax tree modifications with the goal of automatically correcting an exercised defect in the underlying program [16]. A test suite is used to evaluate the effectiveness of each generated abstract syntax tree in preserving desirable behavior and eliminating undesirable behavior.

All of these techniques can take the application outside of its anticipated operating envelope. They therefore have the potential to introduce new anomalies and errors. Of course, ClearView's patches also have the potential to negatively affect the application. Because ClearView performs an ongoing evaluation of each deployed patch, it will quickly discard patches that enable negative effects (such as crashes or attacks) in favor of more effective patches.

# 7. Conclusion

Errors in deployed software systems pose an important threat to the integrity and utility of our computing infrastructure. Relying on manual developer intervention to find and eliminate errors can deny service to application users or even leave the application open to exploitation for long periods of time. ClearView's automatic error detection and correction techniques can provide, with no human intervention whatsoever, the almost immediate correction of errors, including errors that enable newly released security attacks. The result is an application that is immune to the attack and can continue to provide uninterrupted service.

ClearView is targeted toward applications with high availability requirements, for which a small chance of unexpected behavior is preferable to the certainty of denial of service. The feasibility of our approach has been established by a hostile Red Team evaluation in which ClearView automatically patched security vulnerabilities without introducing new attack vectors that the Red Team could exploit. We acknowledge that there are important reliability and security problems that are outside the scope of ClearView. Nevertheless, ClearView addresses an important and realistic problem, and holds out the promise of substantially improving the integrity and availability of our computing infrastructure.

**Acknowledgments.** This work was done while all authors except Larsen and Amarasinghe were at MIT. The research was supported under DARPA cooperative agreement FA8750-06-2-0189 "Collaborative Learning for Security and Repair in Application Communities". We thank our DARPA program manager, Lee Badger, for his enthusiastic support and encouragement. Lee's vision and will-

ingness to commit a significant amount of resources to investigate a new and controversial approach were absolutely critical to the overall success of the project. We thank the outside evaluation team, Cordell Green, Hilarie Orman, and Alex Orso, for their feedback on the project. We thank the White Team, Chris Doh and Dave Kepler, for their fair and impartial management of the Red Team exercise. We thank the hostile Red Team, led by Gregg Tally, for their contributions to the evaluation of ClearView. The Red Team exploit development team included Suresh Krishnaswamy, Robert Lyda, Peter Lovell, David Sames, Richard Toren, and Wayne Morrison. The Red Team negative effect evaluation team included Dave Balenson, Walt Coleman, Vijaya Ramamurthi, and Richard Toren. We thank Andreas Zeller and Andrzej Wasylkowski for their comments on a draft of this paper.

This project was originally structured as a collaboration between MIT and Determina. We thank Sandy Wilbourn, Derek Bruening, Vladimir Kiriansky, Rishi Bidarkar, and Bharath Chandramohan of Determina for their contributions to and support of this project. During the project, VMware acquired Determina. We would to thank Ophir Rachman and Julia Austin of VMware for generously continuing to support this project after the acquisition.

# 8. References

[1] ANAGNOSTAKIS, K., SIDIROGLOU, S., AKRITIDIS, P., XINIDIS, K., MARKATOS, E., AND KEROMYTIS, A. D. Detecting targeted attacks using shadow honeypots. In *USENIX Security* (Aug. 2005).

[2] AUSTIN, T., BREACH, S., AND SOHI, G. Efficient detection of all pointer and array access errors. In *PLDI* (June 2004).

[3] BERGER, E., AND ZORN, B. DieHard: probabilistic memory safety for unsafe languages. In *PLDI* (June 2006).

[4] BRUENING, D. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph.D., MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, Sep. 2004.

[5] CANDEA, G., AND FOX, A. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *HotOS* (Schloss Elmau, Germany, May 2001).

[6] CONDIT, J., HARREN, M., MCPEAK, S., NECULA, G. C., AND WEIMER, W. CCured in the real world. In *PLDI* (June 2003).

[7] COSTA, M., CASTRO, M., ANTONY, ZHOU, L., ZHANG, L., AND PEINADO, M. Bouncer: securing software by blocking bad input. In *SOSP* (Oct. 2007).

[8] COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L., AND BARHAM, P. Vigilante: End-to-End Containment of Internet Worms. In *SOSP* (Oct. 2005).

[9] COWAN, C., PU, C., MAIER, D., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., ZHANG, Q., AND HINTON, H. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX Security* (January 1998).

[10] CRANDALL, J., AND CHONG, F. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *MICRO* (Dec. 2004).

[11] CUI, W., PEINADO, M., WANG, H. J., AND LOCASTO, M. E. ShieldGen: Automatic Data Patch Generation for Unknown Vulnerabilities with Informed Probing. In *IEEE S&P* (May 2007).

[12] DEMSKY, B., ERNST, M. D., GUO, P. J., MCCAMANT, S., PERKINS, J. H., AND RINARD, M. Inference and enforcement of data structure consistency specifications. In *ISSTA* (July 2006).

[13] DEMSKY, B., AND RINARD, M. Data structure repair using goal-directed reasoning. In *ICSE* (May 2005).

[14] ERNST, M. D., PERKINS, J. H., GUO, P. J., MCCAMANT, S., PACHECO, C., TSCHANTZ, M. S., AND XIAO, C. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming 69*, 1–3 (Dec. 2007), 35–45.

[15] ERNST, M. D., PERKINS, J. H., GUO, P. J., MCCAMANT, S., PACHECO, C., TSCHANTZ, M. S., AND XIAO, C. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming 69*, 1–3 (Dec. 2007).

[16] FORREST, S., WEIMER, W., NGUYEN, T., AND GOUES, C. L. A genetic programming approach to automated software repair. In *GECCO* (July 2009).

[17] GRAY, J., AND REUTER, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[18] JIM, T., MORRISETT, G., GROSSMAN, D., HICKS, M., CHENEY, J., AND WANG, Y. Cyclone: A safe dialect of C. In *USENIX* (June 2002).

[19] JONES, R., AND KELLY, P. Backwards-compatible bounds checking for arrays and pointers in C programs. In *AADEBUG* (May 1997).

[20] KENDALL, S. C. Bcc: run-time checking for C programs. In *USENIX Summer* (1983).

[21] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. Secure Execution via Program Shepherding. In *USENIX Security* (Aug. 2002).

[22] LITZKOW, M., AND SOLOMON, M. The evolution of condor checkpointing. In *Mobility: processes, computers, and agents* (1999), ACM Press/Addison-Wesley.

[23] LOCASTO, M. E., SIDIROGLOU, S., AND KEROMYTIS, A. D. Software self-healing using collaborative application communities. In *SNDSS* (Feb. 2005).

[24] NEWSOME, J., AND SONG, D. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *NDSS* (Feb. 2005).

[25] NOVARK, G., BERGER, E., AND ZORN, B. Exterminator: Automatically correcting memory errors with high probability. *Communications of the ACM 51*, 12 (Dec. 2008).

[26] QIN, F., TUCEK, J., SUNDARESAN, J., AND ZHOU, Y. Rx: treating bugs as allergies—a safe method to survive software failures. *SIGOPS Oper. Syst. Rev. 39*, 5 (2005), 235–248.

[27] RINARD, M. Acceptability-oriented computing. In *OOPSLA Companion* (Oct. 2003).

[28] RINARD, M., CADAR, C., DUMITRAN, D., ROY, D. M., AND LEU, T. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *ACSAC* (Dec. 2004).

[29] RINARD, M., CADAR, C., DUMITRAN, D., ROY, D. M., LEU, T., AND WILLIAM S. BEEBEE, J. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *OSDI* (December 2004).

[30] RINARD, M., CADAR, C., AND NGUYEN, H. H. Exploring the acceptability envelope. In *OOPSLA Companion* (Oct. 2005).

[31] RUWASE, O., AND LAM, M. S. A Practical Dynamic Buffer Overflow Detector. In *NDSS* (February 2004).

[32] SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-H., MODADUGU, N., AND BONEH, D. On the Effectiveness of Address-Space Randomization. In *ACM CCS* (Oct. 2004).

[33] SIDIROGLOU, S., GIOVANIDIS, G., AND KEROMYTIS, A. D. A dynamic mechanism for recovering from buffer overflow attacks. In *ISC* (Sep. 2005).

[34] SIDIROGLOU, S., LAADAN, O., KEROMYTIS, A. D., AND NIEH, J. Using rescue points to navigate software recovery. In *IEEE S&P* (May 2007).

[35] SIDIROGLOU, S., LAADAN, O., PEREZ, C., VIENNOT, N., NIEH, J., AND KEROMYTIS, A. D. Assure: automatic software self-healing using rescue points. In *ASPLOS '09* (2009).

[36] SIDIROGLOU, S., LOCASTO, M. E., BOYD, S. W., AND KEROMYTIS, A. D. Building a reactive immune system for software services. In *USENIX* (Apr. 2005).

[37] SMIRNOV, A., AND CHIUEH, T. DIRA: Automatic Detection, Identification and Repair of Control-Hijacking Attacks. In *NDSS* (Feb. 2005).

[38] SPITZNER, L. *Honeypots: Tracking Hackers*. Addison-Wesley, 2002.

[39] Stackshield. `www.angelfire.com/sk/stackshield`.

[40] SUH, G., LEE, J., ZHANG, D., AND DEVADAS, S. Secure Program Execution via Dynamic Information Flow Tracking. In *ASPLOS* (Oct. 2004).

[41] Symantech internet security threat report. `www.symantec.com`, Sep. 2006.

[42] TUCEK, J., NEWSOME, J., LU, S., HUANG, C., XANTHOS, S., BRUMLEY, D., ZHOU, Y., AND SONG, D. Sweeper: A lightweight end-to-end system for defending against fast worms. In *EuroSys* (Mar. 2007).

[43] YONG, S. H., AND HORWITZ, S. Protecting C programs from attacks via invalid pointer dereferences. In *ESEC/FSE* (2003).