# Self-defending software:

# Automatically patching errors in deployed software

Michael Ernst

University of Washington

Joint work with:

Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Sung Kim, Samuel Larsen, Carlos Pacheco, Jeff Perkins, Martin Rinard, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin

# Problem: Your code has bugs and vulnerabilities

- Attack detectors exist
  - Code injection, memory errors (buffer overrun)
- Reaction:
  - Crash the application
    - Loss of data
    - Overhead of restart
    - Attack recurs
    - Denial of service
  - Automatically patch the application

# ClearView:
# Security for legacy software

Requirements:

1. Protect against unknown vulnerabilities
2. Preserve functionality
3. Commercial & legacy software

# 1. Unknown vulnerabilities

- Proactively prevent attacks via unknown vulnerabilities
  - “Zero-day exploits”
  - No pre-generated signatures
  - No hard-coded fixes
  - No time for human reaction
  - Works for bugs as well as attacks

# 2. Preserve functionality

- Maintain continuity: application continues to operate despite attacks
- For applications that require high availability
  - Important for mission-critical applications
  - Web servers, air traffic control, communications
- Technique: create a patch (repair the application)
  - Patching is a valuable option for your toolbox

# 3. Commercial/legacy software

- No modification to source or executables
- No cooperation required from developers
  - Cannot assume built-in survivability features
  - No source information (no debug symbols)
- x86 Windows binaries

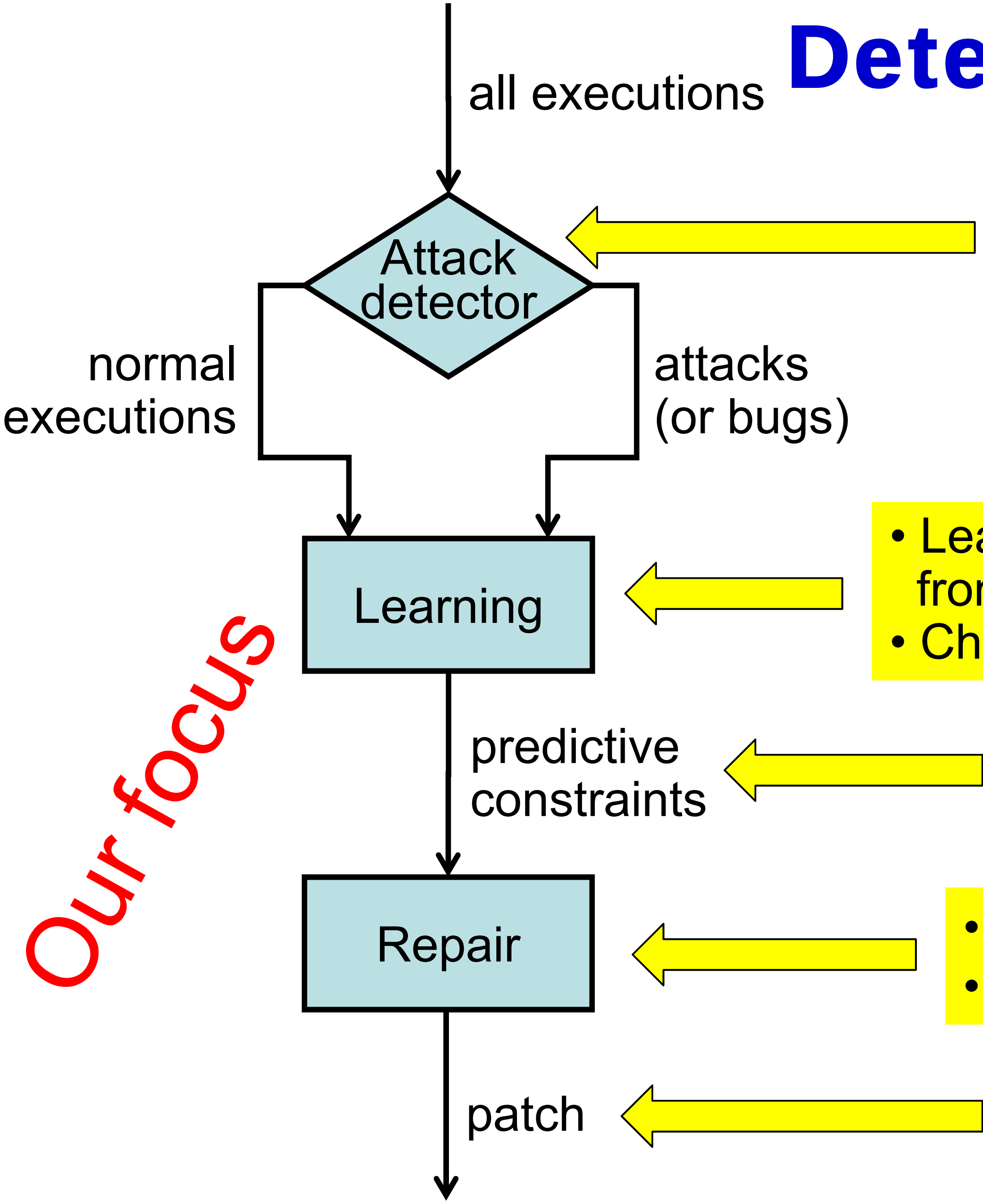
# Learn from success and failure

- Normal executions show what the application is supposed to do

- Each attack (or failure) provides information about the underlying vulnerability
- Repairs improve over time
  - Eventually, the attack is rendered harmless
  - Similar to an immune system
- Detect all attacks (of given types)
  - Prevent negative consequences
  - First few attacks may crash the application

# Detect, learn, repair

[Lin & Ernst 2003]




Pluggable detector,
does not depend on learning

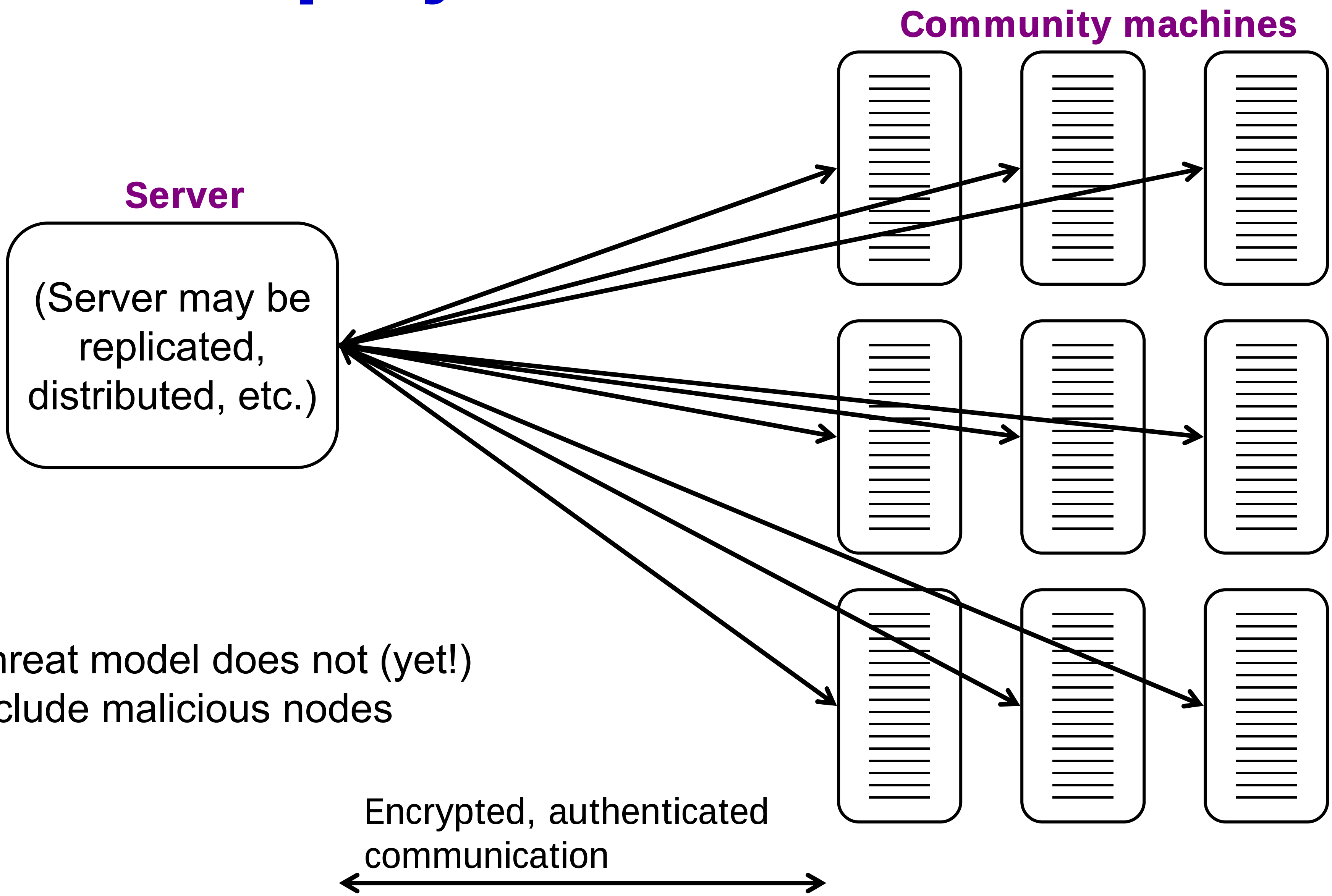
- Learn normal behavior (constraints) from successful runs
- Check constraints during attacks

- True on every good run
- False during every attack

- Patch to re-establish constraints
- Evaluate and distribute patches

Restores normal behavior

# A deployment of ClearView

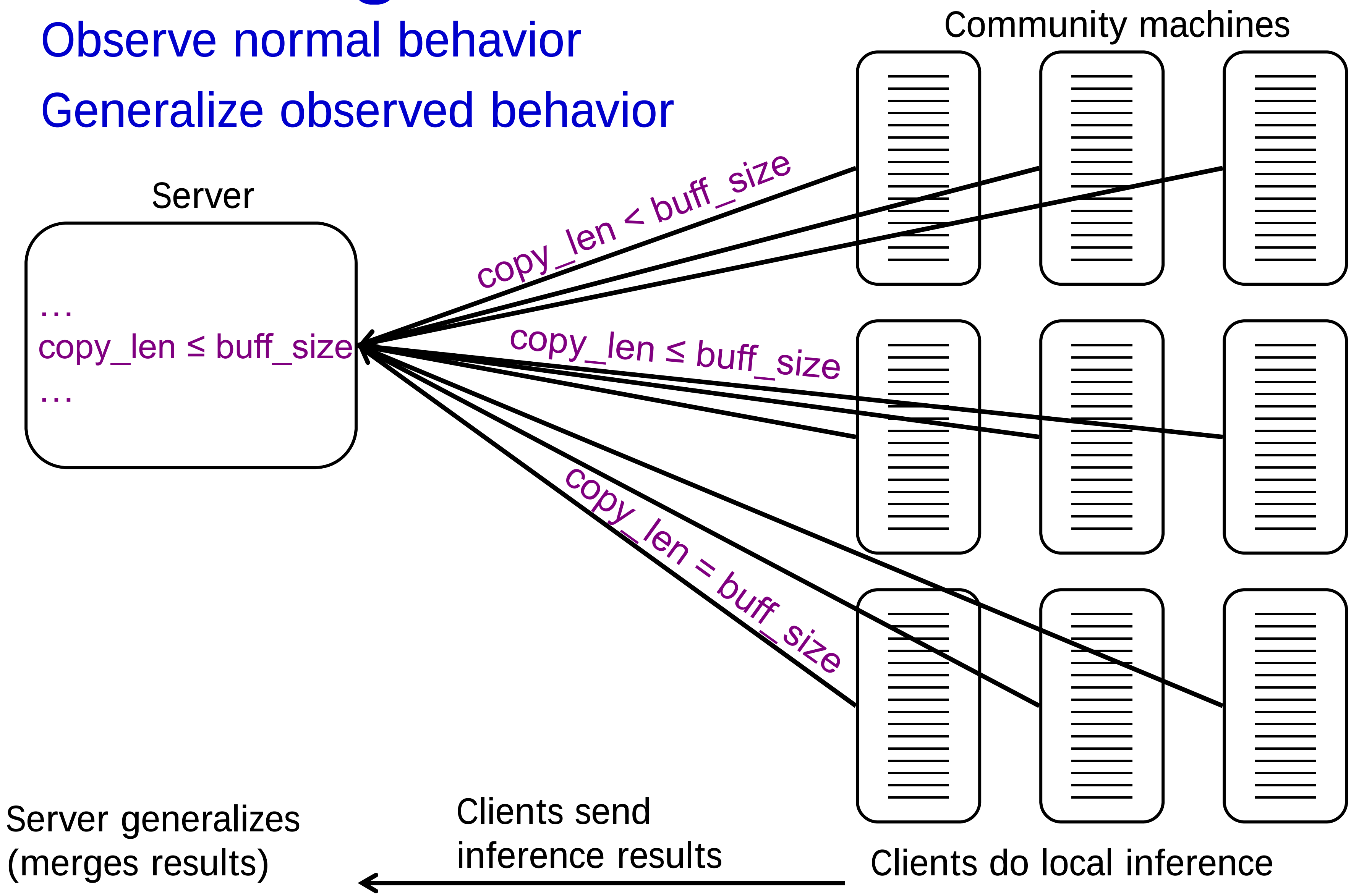



Threat model does not (yet!) include malicious nodes

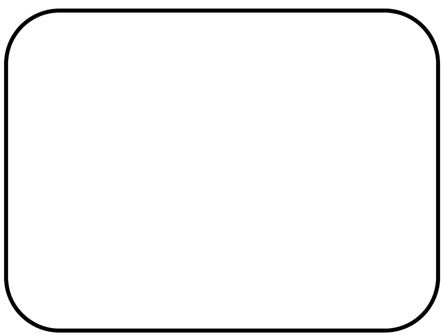
# Learning normal behavior

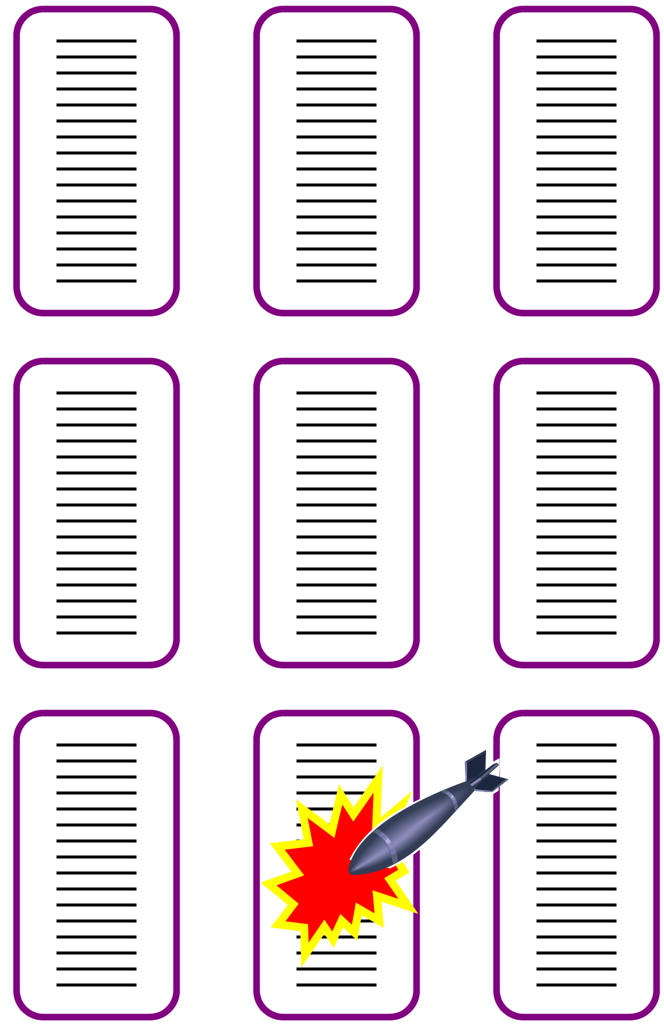
Observe normal behavior

Generalize observed behavior



# Attack detection & suppression



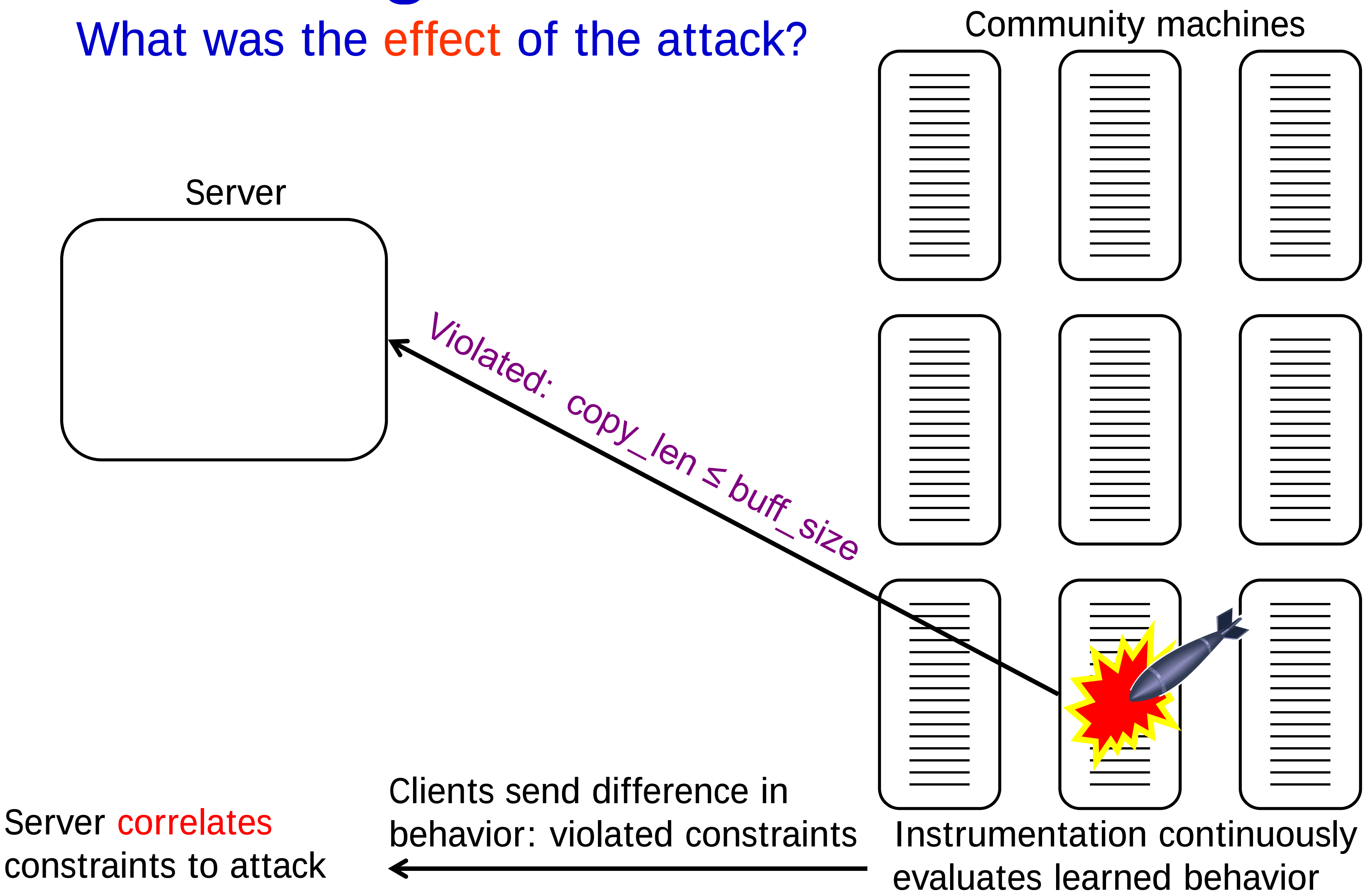

Detectors used in our research:

- Code injection (Memory Firewall)
- Memory corruption (Heap Guard)

Many other possibilities exist

# Learning attack behavior

What was the effect of the attack?



# Repair

Propose a set of patches for each behavior that predicts the attack

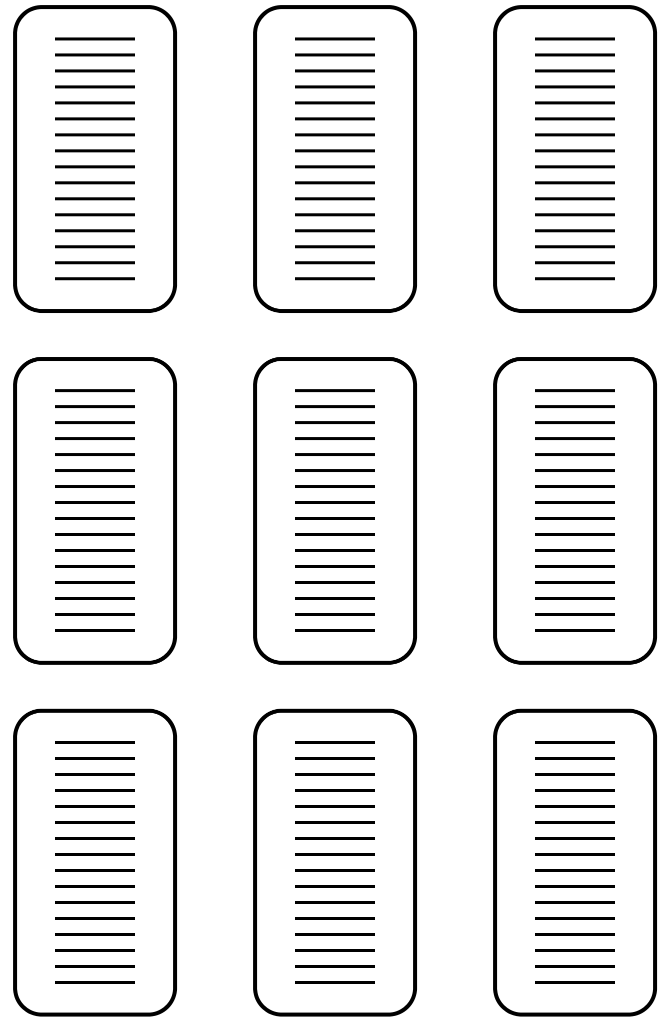
Server

Predictive: copy_len ≤ buff_size

Candidate patches:

1. Set copy_len = buff_size
2. Set copy_len = 0
3. Set buff_size = copy_len
4. Return from procedure

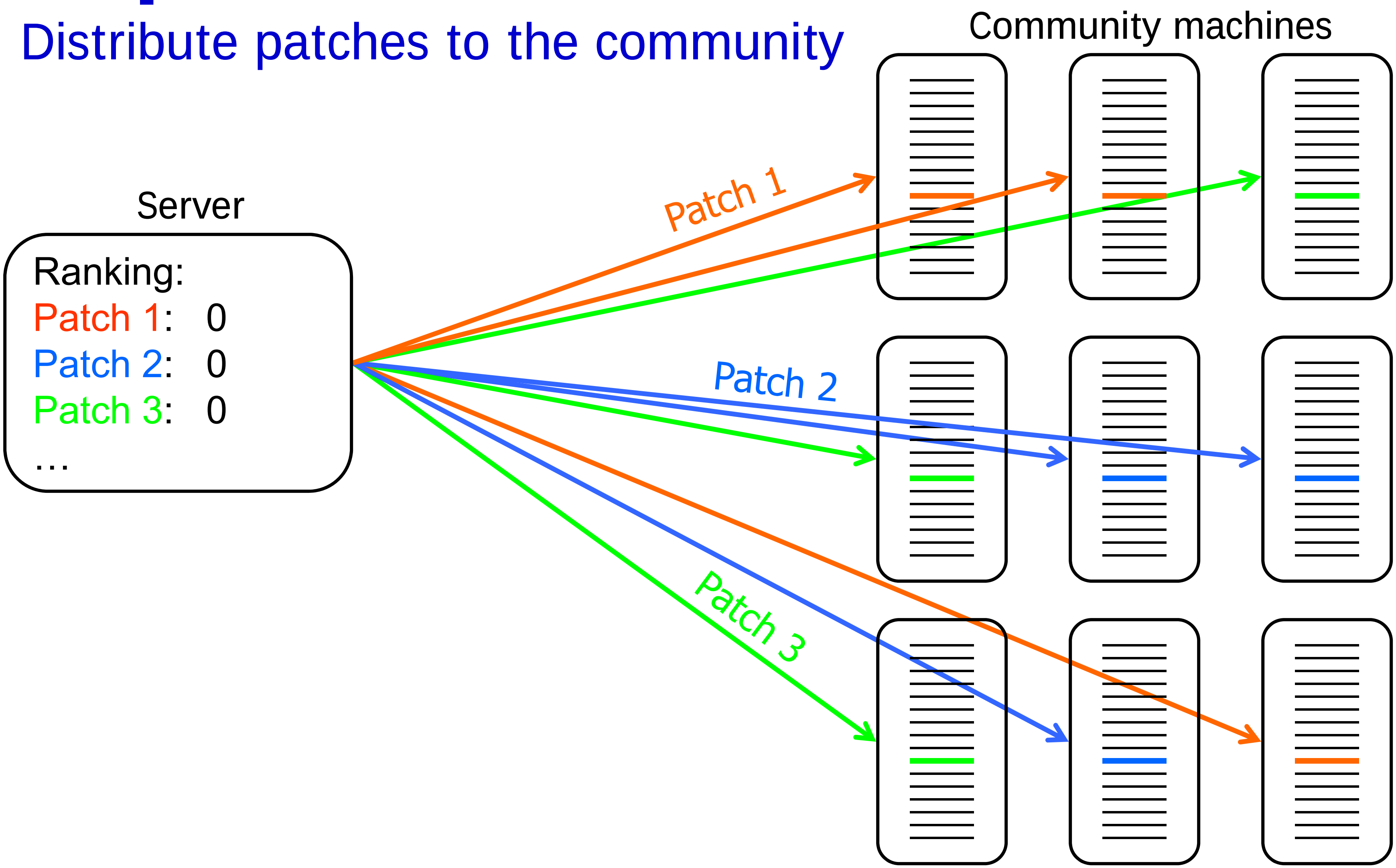
Community machines



Server generates a set of patches

# Repair

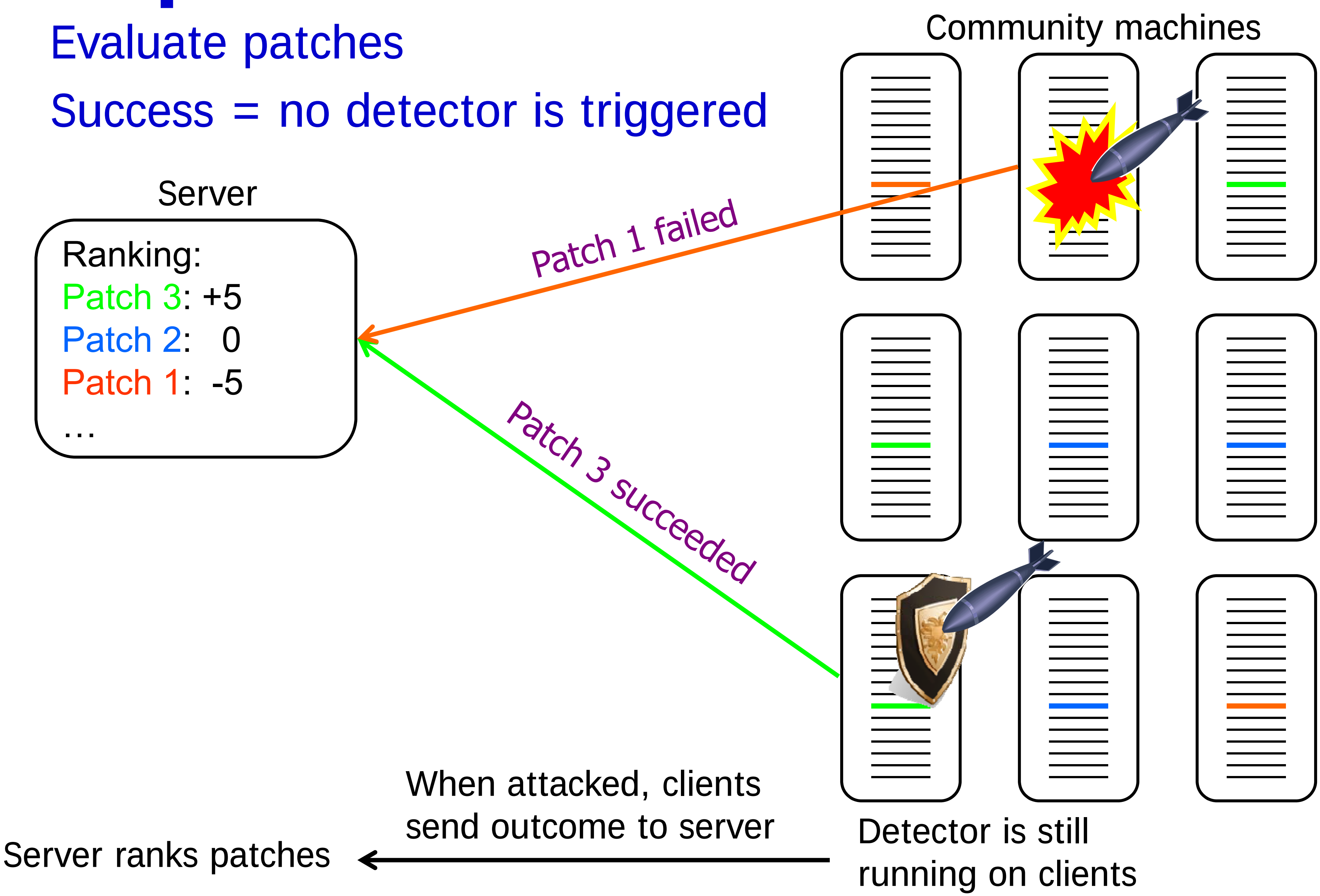
Distribute patches to the community



# Repair

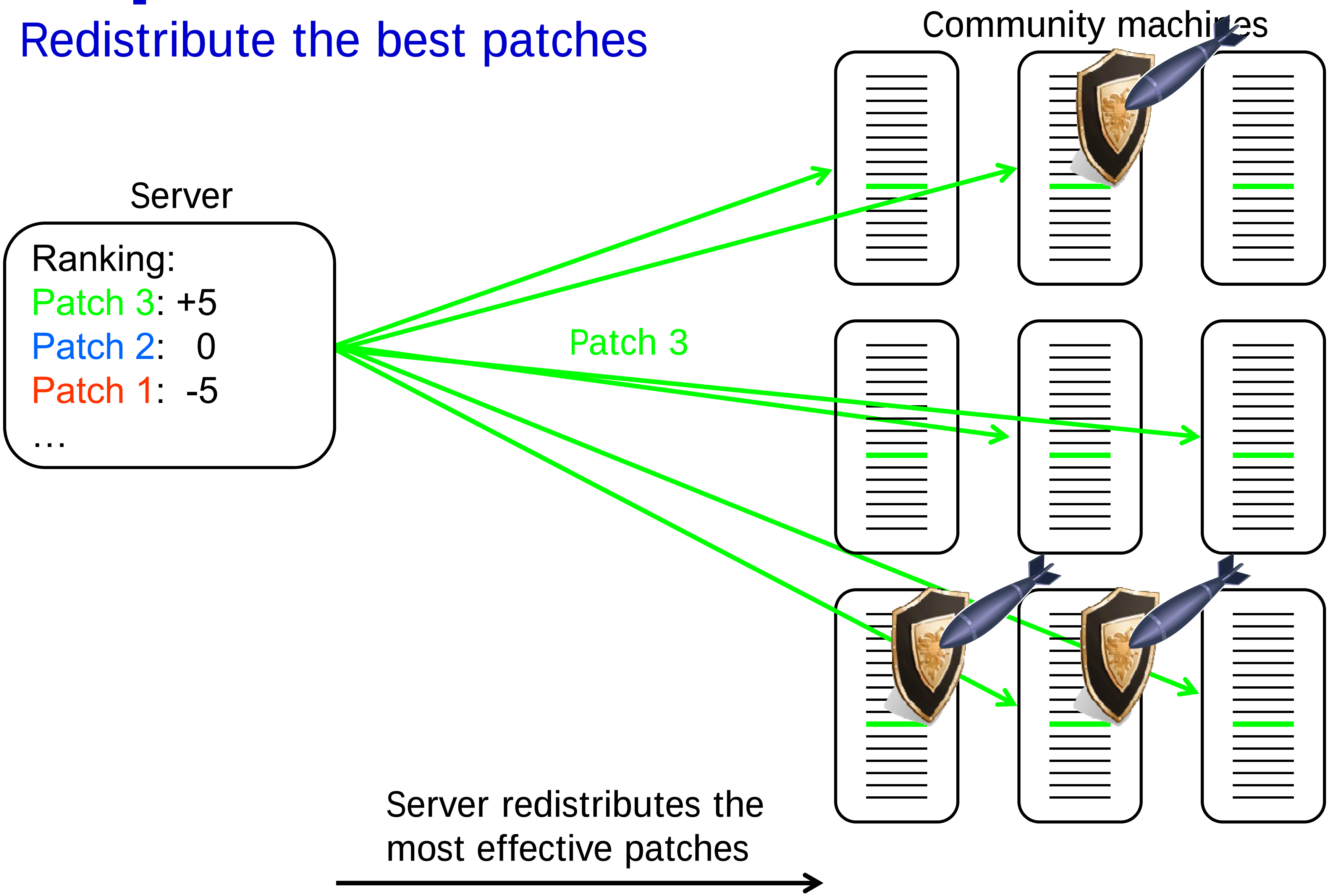
Evaluate patches

Success = no detector is triggered



# Repair

## Redistribute the best patches

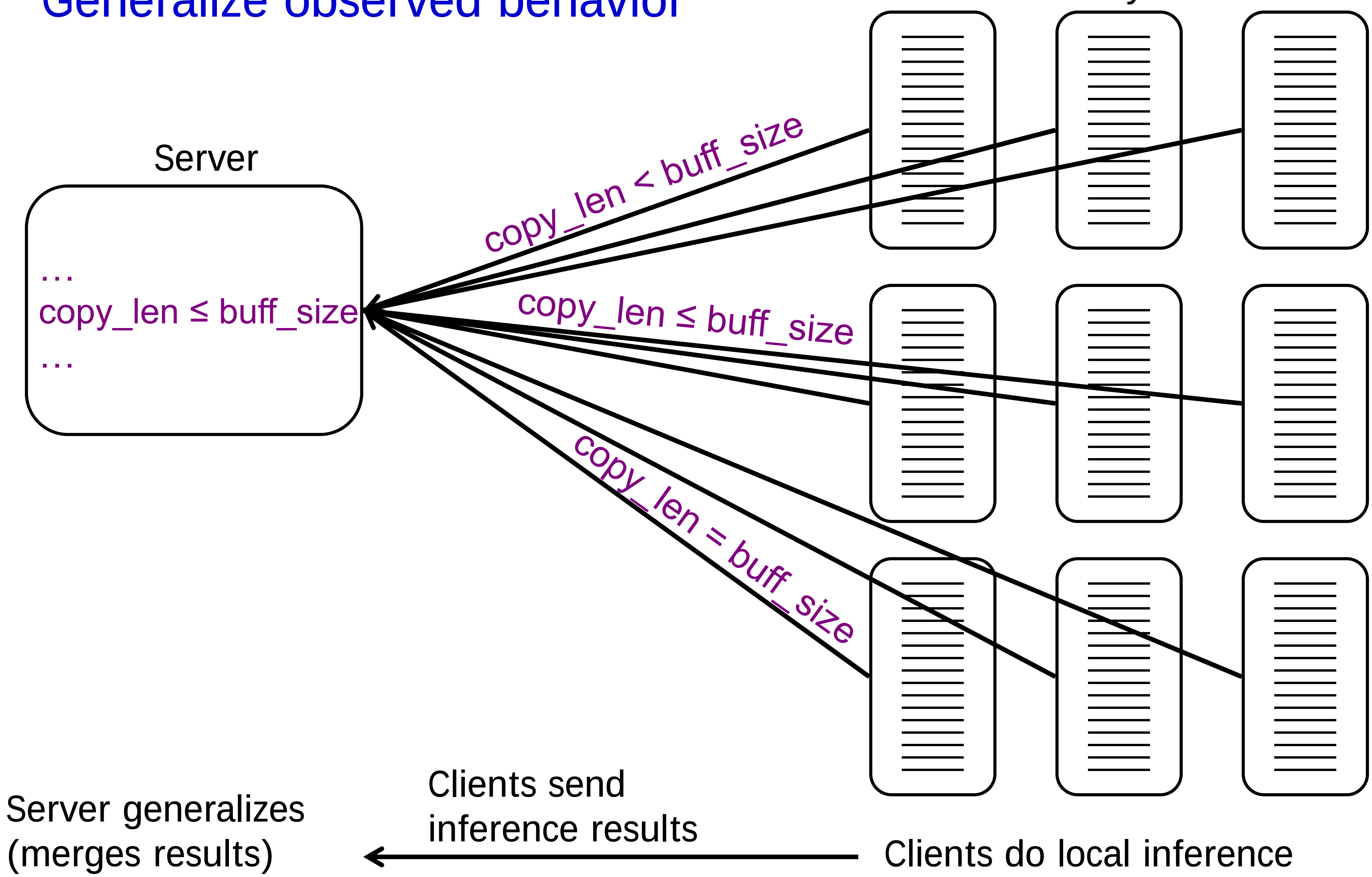


# Outline

- Overview
- Learning normal behavior
- Learning attack behavior
- Repair:  propose and evaluate patches
- Evaluation:  adversarial Red Team exercise
- Conclusion

# Learning normal behavior

Generalize observed behavior



# Dynamic invariant detection

- Daikon generalizes observed program executions

Candidate constraints:

copy_len < buff_size
copy_len ≤ buff_size
copy_len = buff_size
copy_len ≥ buff_size
copy_len > buff_size
copy_len ≠ buff_size

Observation:

copy_len: 22
buff_size: 42

Remaining candidates:

copy_len < buff_size
copy_len ≤ buff_size
~~copy_len = buff_size~~
~~copy_len ≥ buff_size~~
~~copy_len > buff_size~~
copy_len ≠ buff_size

- Many optimizations for accuracy and speed
  - Data structures, code analysis, statistical tests, …
- We further enhanced the technique

# Quality of inference results

- Not sound
  - Overfitting if observed executions are not representative
- Not complete
  - Templates are not exhaustive
- Useful!
- Unsoundness is not a hindrance
  - Does not affect attack detection
  - For repair, mitigated by the correlation step
  - Continued learning improves results

# Outline

- Overview
- Learning normal behavior
- Learning attack behavior
- Repair: propose and evaluate patches
- Evaluation: adversarial Red Team exercise
- Conclusion

# Detecting attacks (or bugs)

Goal: detect problems close to their source

Code injection (Determina Memory Firewall)

- Triggers if control jumps to code that was not in the original executable

Memory corruption (Heap Guard)

- Triggers if sentinel values are overwritten

These have low overhead and no false positives

Other detectors are possible

# Learning from failures

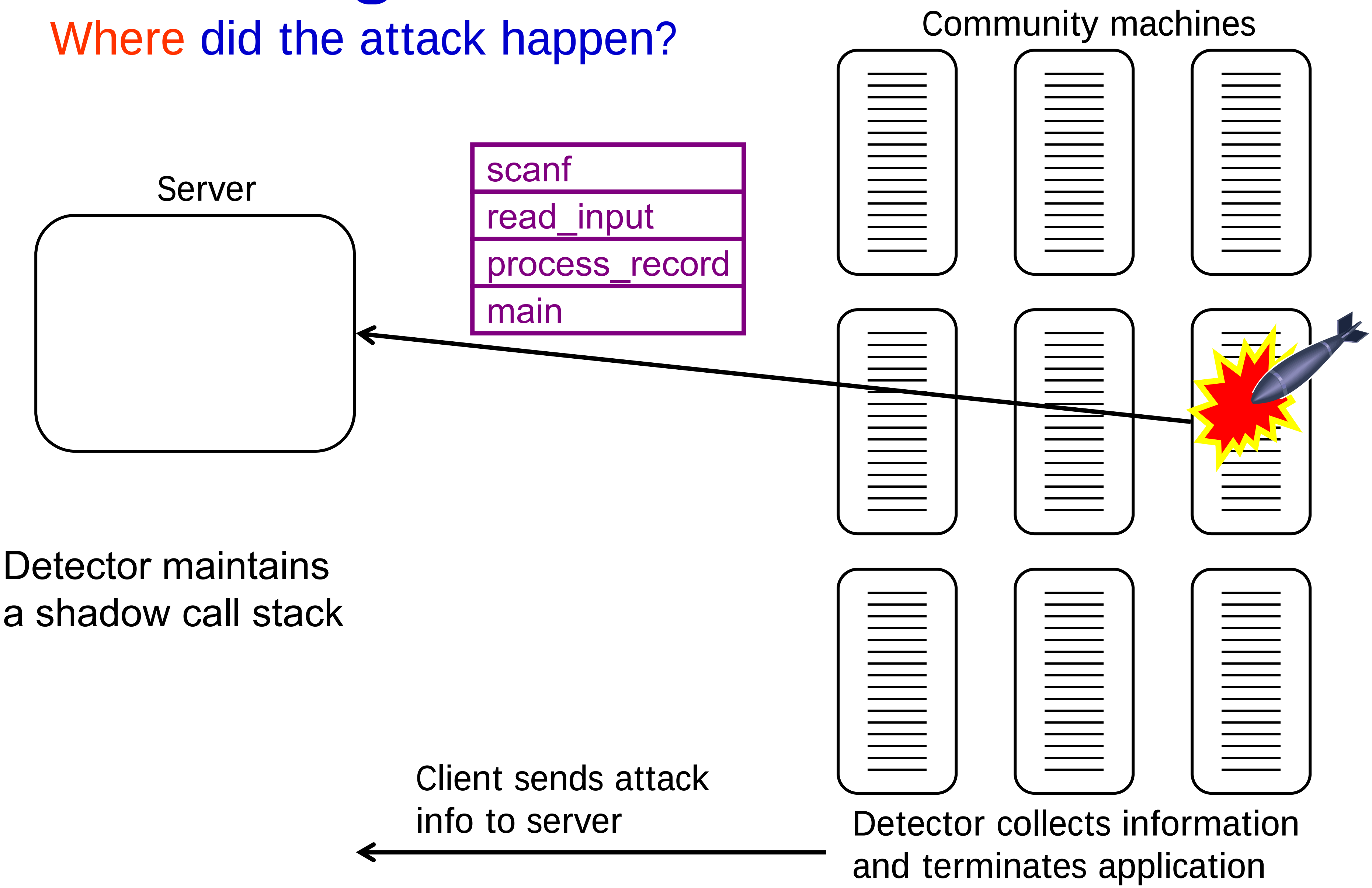
Each attack provides information about the underlying vulnerability

- That it exists
- Where it can be exploited
- How the exploit operates
- What repairs are successful

# Attack detection & suppression

# Learning attack behavior

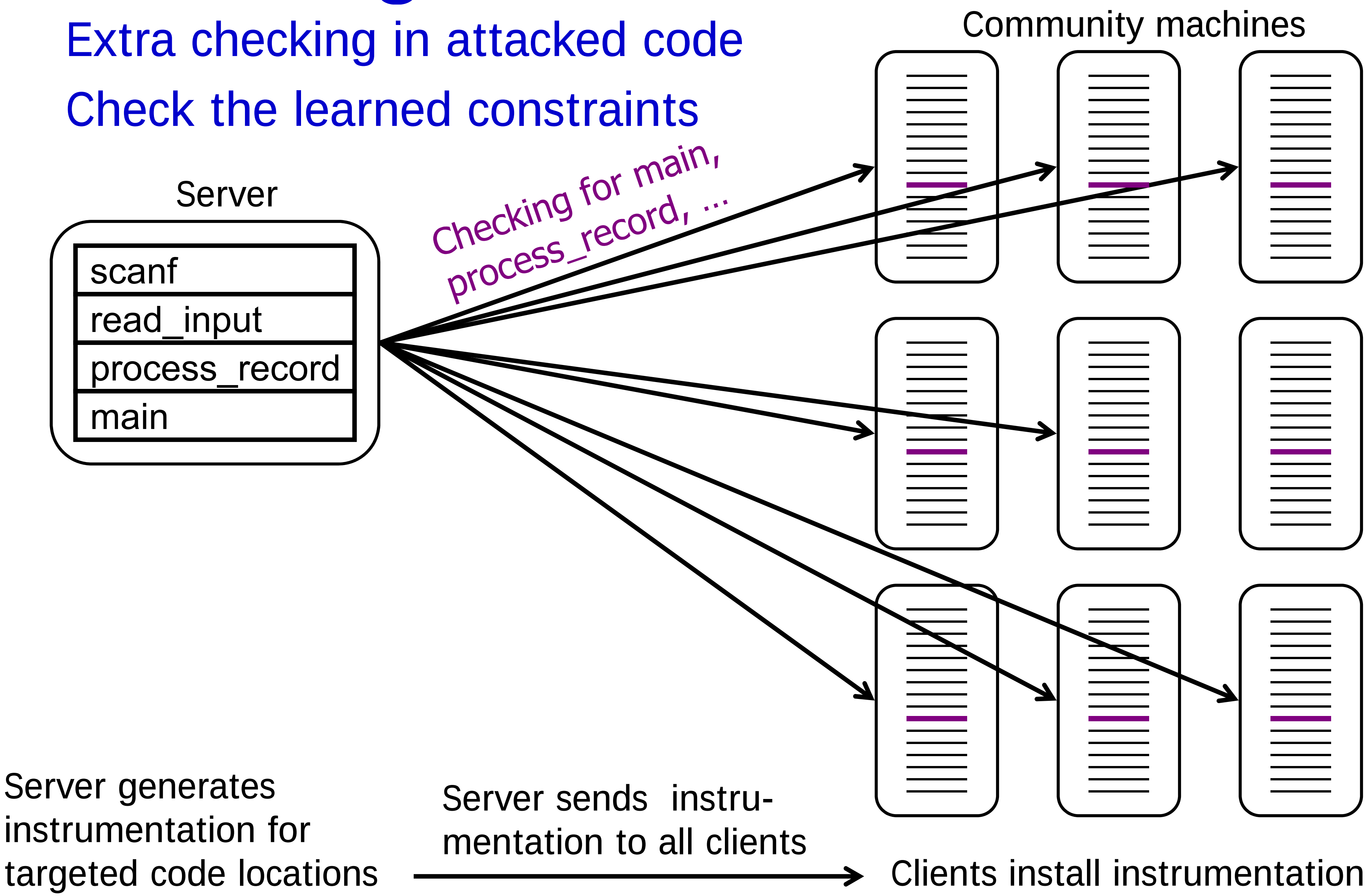
Where did the attack happen?



# Learning attack behavior

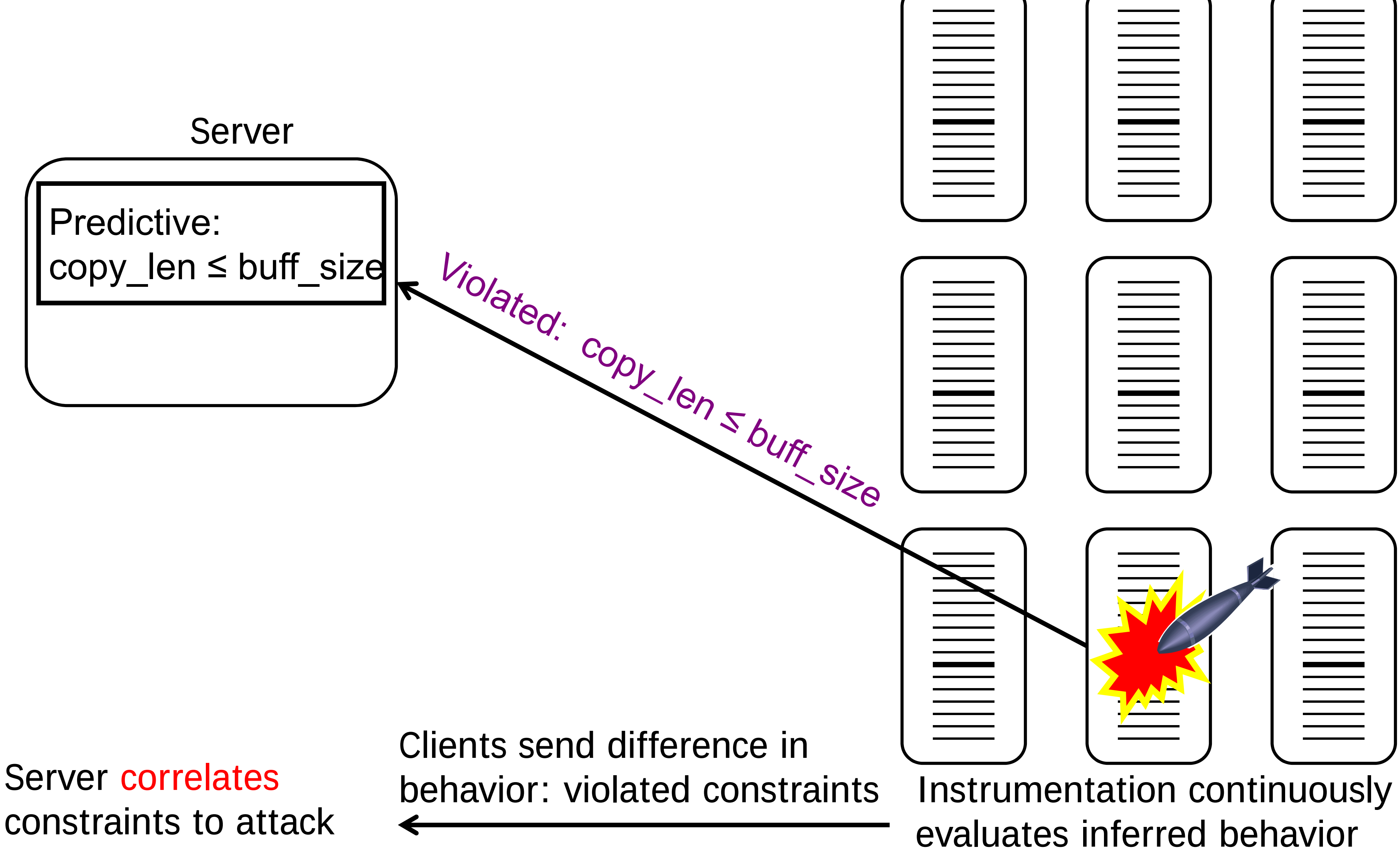
Extra checking in attacked code

Check the learned constraints



# Learning attack behavior

## What was the effect of the attack?



# Correlating attacks & constraints

Check constraints only at attack sites

– Low overhead

A constraint is predictive of an attack if:

– The constraint is violated iff the attack occurs

Create repairs for each predictive constraint

– Re-establish normal behavior

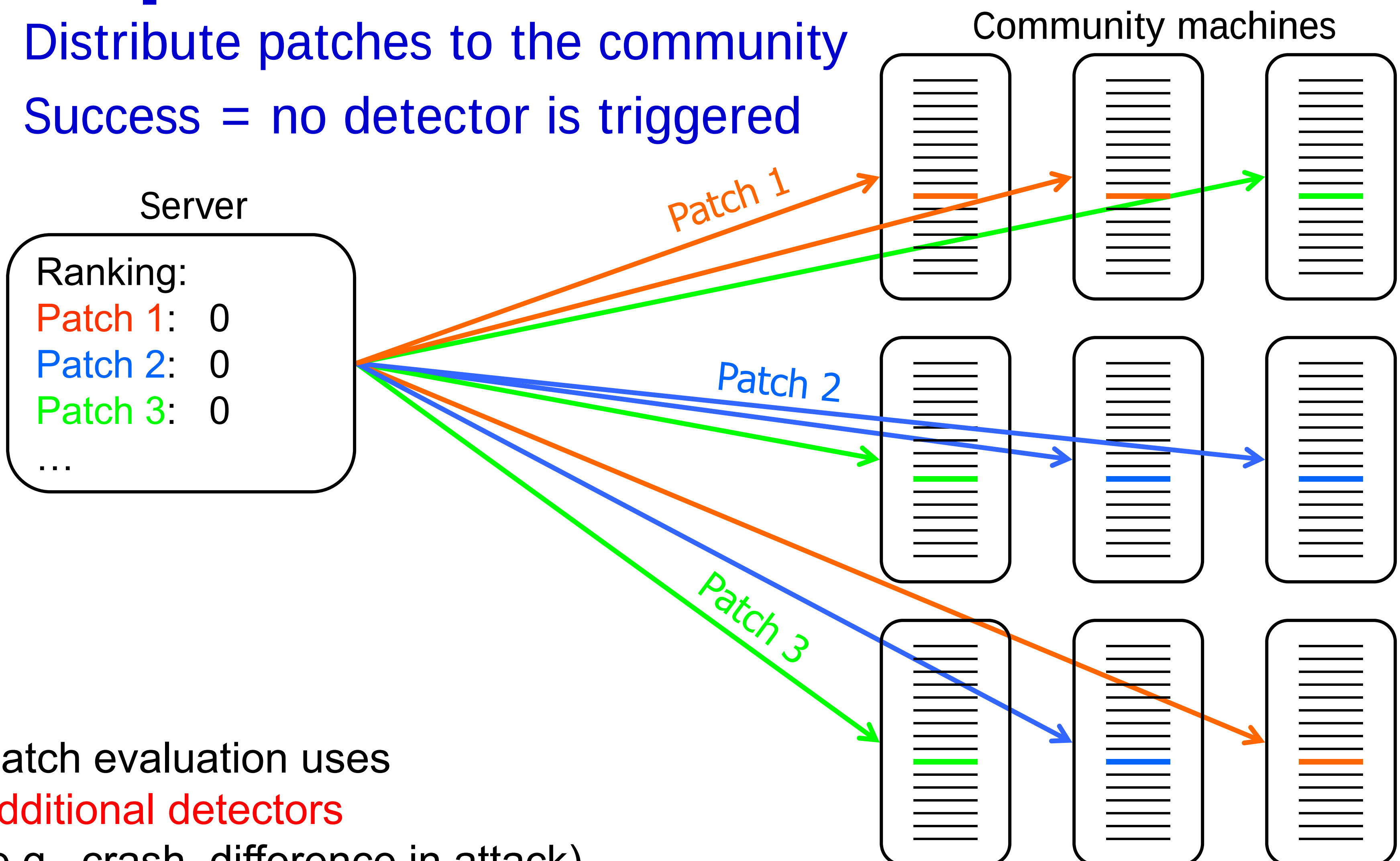
# Outline

- Overview
- Learning normal behavior
- Learning attack behavior
- Repair:  propose and evaluate patches
- Evaluation:  adversarial Red Team exercise
- Conclusion

# Repair

Distribute patches to the community

Success = no detector is triggered




Patch evaluation uses
additional detectors
(e.g., crash, difference in attack)

# Attack example

- Target: JavaScript system routine (written in C++)
  - Casts its argument to a C++ object, calls a virtual method
  - Does not check type of the argument
- Attack supplies an “object” whose virtual table points to attacker-supplied code
- Predictive constraint at the method call:
  - JSRI address target is one of a known set
- Possible repairs:
  - Call one of the known valid methods
  - Skip over the call
  - Return early

# Repair example

```
if (! (copy_len ≤ buff_size))
   copy_len = buff_size;
```

- The repair checks the predictive constraint
  - If constraint is not violated, no need to repair
  - If constraint is violated, an attack is (probably) underway
- The patch does not depend on the detector
  - Should fix the problem before the detector is triggered
- Repair is not identical to what a human would write
  - Unacceptable to wait for human response

# Example constraints & repairs

$v_1 \leq v_2$

```
if (!(v₁≤v₂)) v₁ = v₂;
```

$v \geq c$

```
if (!(v≥c)) v = c;
```

$v \in \{ c_1, c_2, c_3 \}$

```
if (!(v==c₁ || v==c₂ || v==c₃)) v = cᵢ;
```

Return from enclosing procedure

```
if (!(…)) return;
```

Modify a use: convert "**`call *v`**" to

```
if (…) call *v;
```

Constraint on **`v`** (not negated)

# Evaluating a patch

- In-field evaluation
  - No attack detector is triggered
  - No other behavior deviations
    - E.g., crash, application invariants
- Pre-validation, before distributing the patch:
  - Replay the attack
    + No need to wait for a second attack
    + Exactly reproduce the problem
    – Expensive to record log; log terminates abruptly
    – Need to prevent irrevocable effects
    – Delays distribution of good patches
  - Run the program's test suite
    – May be too sensitive
    – Not available for commercial software

# Outline

- Overview
- Learning normal behavior
- Learning attack behavior
- Repair:  propose and evaluate patches
- Evaluation:  adversarial Red Team exercise
- Conclusion

# Red Team

- Red Team attempts to break our system
  - Hired by DARPA; 10 engineers
- Red Team created 10 Firefox exploits
  - Each exploit is a webpage
  - Firefox executes arbitrary code
  - Malicious JavaScript, GC errors, stack smashing, heap buffer overflow, uninitialized memory

# Rules of engagement

- Firefox 1.0
  - ClearView may not be tuned to known vulnerabilities
  - Focus on most security-critical components
    - No access to a community for learning
- Red Team has access to all ClearView materials
  - Source code, documents, learned invariants, …

# ClearView was successful

- Detected all attacks, prevented all exploits
- For 7/10 vulnerabilities, generated a patch that maintained functionality
  - No observable deviation from desired behavior
  - After an average of 4.9 minutes and 5.4 attacks
- Handled polymorphic attack variants
- Handled simultaneous & intermixed attacks
- No false positives
- Low overhead for detection & repair

# 3 un-repaired vulnerabilities

Consequence: Application crashes when attacked. No exploit occurs.

1. ClearView was mis-configured: didn't try repairs in all procedures on the stack
2. Learning suite was too small: a needed constraint was not statistically significant
3. A needed constraint was not built into Daikon

# Outline

- Overview
- Learning normal behavior
- Learning attack behavior
- Repair:  propose and evaluate patches
- Evaluation:  adversarial Red Team exercise
- Conclusion

# Limitations

ClearView might fail to repair an error:

- Only fixes errors for which a detector exists
- Daikon might not learn a needed constraint
- Predictive constraint may be too far from error
- Built-in repairs may not be sufficient

ClearView might degrade the application:

- Patch may impair functionality
- Attacker may subvert patch
- Malicious nodes may induce bad patches

Bottom line: Red Team tried unsuccessfully

# Related work

- Attack detection:  ours are mostly standard
  - Distributed:  Vigilante [Costa], live monitoring [Kıcıman], statistical bug isolation [Liblit]
- Learning
  - FSMs of system calls for anomaly detection
  - Invariants: [Lin], [Demsky], Gibraltar [Baliga]
  - System configuration:  FFTV [Lorenzoli], Dimmunix [Jula]
- Repair & failure tolerance
  - Checkpoint and replay:  Rx [Qin], microreboot [Candea]
  - Failure-oblivious [Rinard], ASSURE [Sidiroglou]

# Credits

- Saman Amarasinghe
- Jonathan Bachrach
- Michael Carbin
- Michael Ernst
- Sung Kim
- Samuel Larsen
- Carlos Pacheco
- Jeff Perkins
- Martin Rinard
- Frank Sherwood
- Stelios Sidiroglou
- Greg Sullivan
- Weng-Fai Wong
- Yoav Zibin

Subcontractor: Determina, Inc.
Funding: DARPA (PM: Lee Badger)
Red Team: SPARTA, Inc.

# Contributions

ClearView: framework for patch generation

– Pluggable detection, learning, repair

1. Protects against unknown vulnerabilities
   – Learns from success
   – Learns from failure: what, where, how
   – Learning focuses effort where it is needed
2. Preserves functionality: repairs the vulnerability
3. Commercial software: Windows binaries

Evaluation via a Red Team exercise