From STEM to SEAD: Speculative Execution for Automated Defense

Michael E. Locasto, Angelos Stavrou, Gabriela F. Cretu, Angelos D. Keromytis
Department of Computer Science, Columbia University
{locasto, angel, gcretu, angelos}@*cs.columbia.edu*

Abstract

Most computer defense systems crash the process that they protect as part of their response to an attack. Although recent research explores the feasibility of self-healing to automatically recover from an attack, self-healing faces some obstacles before it can protect legacy applications and COTS (Commercial Off–The–Shelf) software. Besides the practical issue of not modifying source code, self-healing must know both when to engage and how to guide a repair.

Previous work on a self-healing system, STEM, left these challenges as future work. This paper improves STEM's capabilities along three lines to provide practical speculative execution for automated defense (SEAD). First, STEM is now applicable to COTS software: it does not require source code, and it imposes a roughly 73% performance penalty on Apache's normal operation. Second, we introduce *repair policy* to assist the healing process and improve the semantic correctness of the repair. Finally, STEM can create behavior profiles based on aspects of data and control flow.

1 Introduction

Most software applications lack the ability to repair themselves during an attack, especially when attacks are delivered via previously unseen inputs or exploit previously unknown vulnerabilities. Self-healing software, an emerging area of research [31, 29, 28], involves the creation of systems capable of automatic remediation of faults and attacks. In addition to detecting and defeating an attack, self-healing systems seek to correct the integrity of the computation itself. Self-healing countermeasures serve as a first line of defense while a slower but potentially more complete human-driven response takes place.

Most self-healing mechanisms follow what we term the ROAR (Recognize, Orient, Adapt, Respond) workflow. These systems (a) *Recognize* a threat or attack has occurred, (b) *Orient* the system to this threat by analyzing it, (c) *Adapt* to the threat by constructing appropriate fixes or changes in state, and finally (d) *Respond* to the threat by verifying and deploying those adaptations.

While embryonic attempts in this space demonstrate the feasibility of the basic concept, these techniques face a few obstacles before they can be deployed to protect and repair legacy systems, production applications, and COTS (Commercial Off–The–Shelf) software. The key challenge is to apply a fix inline (*i.e.*, as the application experiences an attack) without restarting, recompiling, or replacing the process.

Executing through a fault in this fashion involves overcoming several obstacles. First, the system should not make changes to the application's source code. Instead, we supervise execution with dynamic binary rewriting. Second, the semantics of program execution must be maintained as closely as possible to the original intent of the application's author. We introduce *repair policy* to guide the semantics of the healing process. Third, the supervised system may communicate with external entities that are beyond the control or logical boundary of the self-healing system. We explore the design space of *virtual proxies* and detail one particular vector of implementation to address this problem. Finally, the system must employ detection mechanisms that can indicate when to supervise and heal the application's execution. Although STEM can operate with a number of detectors, we show how it gathers aspects of both data and control flow to produce an application's behavior profile.

1.1 Motivation

Our solutions are primarily motivated by the need to address the limitations of our previous self-healing software system prototype [31]. STEM (Selective Transactional EMulation) provides self-healing by speculatively executing "slices" of a process. We based this first approach on a feedback loop that inserted calls to an x86 emulator at vulnerable points in an application's source code (requiring recompilation and redeployment). STEM supervises the application using microspeculation and error virtualization.

1.1.1 Microspeculation and Error Virtualization

The basic premise of our previous work is that portions of an application can be treated as a transaction. Functions serve as a convenient abstraction and fit the transaction role well in most situations [31]. Each transaction (vulnerable code slice) can be speculatively executed in a sandbox environment. In much the same way that a processor speculatively executes past a branch instruction and discards the mispredicted code path, STEM executes the transaction's instruction stream, optimistically "speculating" that the results of these computations are benign. If this *microspeculation* succeeds, then the computation simply carries on. If the transaction experiences a fault or exploited vulnerability, then the results are ignored or replaced according to the particular response strategy being employed. We call one such strategy, as discussed in previous work [31], *error virtualization*.

```
0  int bar(char* buf)
1  {
2    char rbuf[10];
3    int i=0;
4    if(NULL==buf)
5     return -1;
6    while(i<strlen(buf))
7    {
8     rbuf[i++]=*buf++;
9    }
10   return 0;
11 }
```

Figure 1: *Error Virtualization.* We can map unanticipated errors, like an exploit of the buffer overflow vulnerability in line 8, to anticipated error conditions explicitly handled by the existing program code (like the error condition return in line 5).

The key assumption underlying error virtualization is that a mapping can be created between the set of errors that *could* occur during a program's execution and the limited set of errors that the program code explicitly handles. By virtualizing errors, an application can continue execution through a fault or exploited vulnerability by nullifying its effects and using a manufactured return value for the function where the fault occurred. In the previous version of STEM, these return values were determined by source code analysis on the return type of the offending function. Vanilla error virtualization seems to work best with server applications — applications that typically have a request processing loop that can presumably tolerate minor errors in a particular trace of the loop. This paper, however, aims to provide a practical solution for client applications (*e.g.,* email, messaging, authoring, browsing) as well as servers.

1.1.2 Limitations of Previous Approach

Recently proposed approaches to self-healing such as *error virtualization* [31] and *failure-oblivious computing* [29] prevent exploits from succeeding by masking failures. However, error virtualization fails about 12% of the time, and both approaches have the potential for semantically incorrect execution. These shortcomings are devastating for applications that perform precise[1] calculations or provide authentication & authorization.

Furthermore, error virtualization required access to the source code of the application to determine appropriate error virtualization values and proper placement of the calls to the supervision environment. A better solution would operate on unmodified binaries and profile the application's behavior to learn appropriate error virtualization values during runtime.

Finally, as with all systems that rely on rewinding execution [4, 28] after a fault has been detected, I/O with external entities remains uncontrolled. For example, if a server program supervised by STEM writes a message to a network client during microspeculation, there is no way to "take back" the message: the state of the remote client has been irrevocably altered.

1.2 Contributions

The changes we propose and evaluate in this paper provide the basis for the redesign of STEM's core mechanisms as well as the addition of novel methods to guide the semantic correctness of the self-healing response. STEM essentially adds a policy-driven layer of indirection to an application's execution. The following contributions *collectively* provide a significant improvement over previous work:

1. **Eliminate Source-Level Modifications** – We employ error virtualization and microspeculation (and the new techniques proposed in this section) during binary rewriting. STEM serves as a self-contained environment for supervising applications without recompiling or changing source code.

2. **Virtual Proxies** – Self-healing techniques like microspeculation have difficulty "rewinding" the results of communication with remote entities that are not under the control of the self-healing system. This challenge can affect the semantic correctness of the healing process. We examine the notion of

virtual proxies to support cooperative microspeculation without changing the communications protocols or the code of the remote entity.

3. **Repair Policy** – Error virtualization alone is not appropriate for all functions and applications, especially if the function is not idempotent or if the application makes scientific or financial calculations or includes authentication & authorization checks. *Repair policy* provides a more complete approach to managing the semantic correctness of a repair. Section 4 describes the most relevant aspects and key features of repair policy as well as STEM's support for interpreting it. We refer the interested reader to our technical report [20] for a more complete discussion of the theoretical framework.

4. **Behavior Profiling** – Because we implement STEM in a binary supervision environment, we can non-invasively collect a profile of application behavior by observing aspects of both data and control flow. This profile assists in detection (detecting deviations from the profile), repair (selecting appropriate error virtualization values), and repair validation (making sure that future performance matches past behavior).

Using STEM to supervise dynamically linked applications directly from startup incurs a significant performance penalty (as shown in Table 2), especially for short-lived applications. Most of the work done during application startup simply loads and resolves libraries. This type of code is usually executed only once, and it probably does not require protection. Even though it may be acceptable to amortize the cost of startup over the lifetime of the application, we can work around the startup performance penalty by employing some combination of three reasonable measures: (1) statically linking applications, (2) only attaching STEM after the application has already started, (3) delay attaching until the system observes an IDS alert. We evaluate the second option by attaching STEM to Apache after Apache finishes loading. Our results (shown in Table 1) indicate that Apache experiences roughly a 73% performance degradation under STEM.

2 Related Work

Self-healing mechanisms complement approaches that stop attacks from succeeding by preventing the injection of code, transfer of control to injected code, or misuse of existing code. Approaches to automatically defending software systems have typically focused on ways to proactively protect an application from attack. Examples of these proactive approaches include writing the system in a "safe" language, linking the system with "safe" libraries [2], transforming the program with artificial diversity, or compiling the program with stack integrity checking [9]. Some defense systems also externalize their response by generating either vulnerability [8, 24, 10] or exploit [19, 22, 32, 36] signatures to prevent malicious input from reaching the protected system.

2.1 Protecting Control Flow

Starting with the technique of *program shepherding* [17], the idea of enforcing the integrity of control flow has been increasingly researched. Program shepherding validates branch instructions to prevent transfer of control to injected code and to make sure that calls into native libraries originate from valid sources. Control flow is often corrupted because input is eventually incorporated into part of an instruction's opcode, set as a jump target, or forms part of an argument to a sensitive system call. Recent work focuses on ways to prevent these attacks using tainted dataflow analysis [34, 25, 8].

Abadi *et al.* [1] propose formalizing the concept of Control Flow Integrity (CFI), observing that high-level programming often assumes properties of control flow that are not enforced at the machine level. CFI provides a way to statically verify that execution proceeds within a given control-flow graph (the CFG effectively serves as a policy). The use of CFI enables the efficient implementation of a software shadow call stack with strong protection guarantees. CFI complements our work in that it can enforce the invocation of STEM (rather than allowing malcode to skip past its invocation).

2.2 Self-Healing

Most defense mechanisms usually respond to an attack by terminating the attacked process. Even though it is considered "safe", this approach is unappealing because it leaves systems susceptible to the original fault upon restart and risks losing accumulated state.

Some first efforts at providing effective remediation strategies include failure oblivious computing [29], error virtualization [31], rollback of memory updates [32], crash-only software [5], and data structure repair [11]. The first two approaches may cause a semantically incorrect continuation of execution (although the Rx system [28] attempts to address this difficulty by exploring semantically safe alterations of the program's environment). Oplinger and Lam [26] employ hardware Thread-Level Speculation to improve software reliability. They execute an application's monitoring code in parallel with the primary computation and roll back the computation "transaction" depending on the results of the monitoring code. Rx employs proxies that are somewhat akin to our virtual proxies, although Rx's are more powerful in that they explicitly deal with protocol syntax and semantics during replay.

ASSURE [30] is a novel attempt to minimize the likelihood of a semantically incorrect response to a fault or attack. ASSURE proposes the notion of *error virtualization rescue points*. A rescue point is a program location that is known to successfully propagate errors and recover execution. The insight is that a program will respond to malformed input differently than legal input; locations in the code that successfully handle these sorts of anticipated input "faults" are good candidates for recovering to a safe execution flow. ASSURE can be understood as a type of exception handling that dynamically identifies the best scope to handle an error.

2.3 Behavior-based Anomaly Detection

STEM also provides a mechanism to capture aspects of an application's behavior. This profile can be employed for three purposes: (a) to detect application misbehavior, (b) to aid self-healing, and (c) to validate the self-healing response and ensure that the application does not deviate further from its known behavior. STEM captures aspects of both control flow (via the execution context) and portions of the data flow (via function return values). This mechanism draws from a rich literature on host-based anomaly detection.

The seminal work of Hofmeyr, Somayaji, and Forrest [15, 33] examines an application's behavior at the system-call level. Most approaches to host-based intrusion detection perform anomaly detection [6, 13, 14] on sequences of system calls. The work of Feng *et al.* [12] includes an excellent overview of the literature circa 2003. The work of Bhatkar *et al.* [3] also contains a good overview of the more recent literature and offers a technique for *dataflow* anomaly detection to complement traditional approaches that concentrate mostly on control flow. Behavior profiling's logical goal is to create policies for detection [27, 18] and self-healing.

3 STEM

One of this paper's primary contributions is the reimplementation of STEM to make it applicable in situations where source code is not available. This section reviews the technical details of STEM's design and implementation. We built STEM as a tool for the IA-32 binary rewriting PIN [23] framework.

3.1 Core Design

PIN provides an API that exposes a number of ways to instrument a program during runtime, both statically (as a binary image is loaded) and dynamically (as each instruction, basic block, or procedure is executed). PIN tools contain two basic types of functions: (1) instrumentation functions and (2) analysis functions. When a PIN tool starts up, it registers instrumentation functions that serve as callbacks for when PIN recognizes an event or portion of program execution that the tool is interested in (*e.g.*, instruction execution, basic block entrance or exit, *etc.*). The instrumentation functions then employ the PIN API to insert calls to their analysis functions. Analysis functions are invoked every time the corresponding code slice is executed; instrumentation functions are executed only the first time that PIN encounters the code slice.

STEM treats each function as a transaction. Each "transaction" that should be supervised (according to policy) is speculatively executed. In order to do so, STEM uses PIN to instrument program execution at four points: function entry (*i.e.*, immediately before a CALL instruction), function exit (*i.e.*, between a LEAVE and RET instruction), immediately before the instruction *after* a RET executes, and for each instruction of a supervised function that writes to memory. The main idea is that STEM inserts instrumentation at both the start and end of each transaction to save state and check for errors, respectively. If microspeculation of the transaction encounters any errors (such as an attack or other fault), then the instrumentation at the end of the transaction invokes cleanup, repair, and repair validation mechanisms.

STEM primarily uses the "Routine" hooks provided by PIN. When PIN encounters a function that it has not yet instrumented, it invokes the callback instrumentation function that STEM registered. The instrumentation function injects calls to four analysis routines:

1. `STEM_Preamble()` – executed at the beginning of each function.
2. `STEM_Epilogue()` – executed before a RET instruction
3. `SuperviseInstruction()` – executed before each instruction of a supervised function
4. `RecordPreMemWrite()` – executed before each instruction of a supervised function that writes to memory

STEM's instrumentation function also intercepts some system calls to support the "CoSAK" supervision policy (discussed below) and the virtual proxies (discussed in Section 5).

3.2 Supervision Policy

One important implementation tradeoff is whether the decision to supervise a function is made at injection time (*i.e.* during the instrumentation function) or at analysis time (*i.e.*, during an analysis routine). Consulting policy and making a decision in the latter (as the current implementation does) allows STEM to change the coverage supervision policy (that is, the set of functions it monitors) during runtime rather than needing to restart the application. Making the decision during injection time is possible, but not for all routines, and since the policy decision is made only once, the set of functions that STEM can instrument is not dynamically adjustable unless the application is restarted, or PIN removes all instrumentation and invokes instrumentation for each function again.

Therefore, each injected analysis routine determines dynamically if it should actually be supervising the current function. STEM instructs PIN to instrument *all* functions – a STEM analysis routine needs to gain control, even if just long enough to determine it should not supervise a particular function. The analysis routines invoke STEM's `ShouldSuperviseRoutine()` function to check the current supervision coverage policy in effect. Supervision coverage policies dictate which subset of an application's functions STEM should protect. These policies include:

- NONE – no function should be microspeculated
- ALL – all functions should be microspeculated
- RANDOM – a random subset should be microspeculated (the percentage is controlled by a configuration parameter)
- COSAK – all functions within a call stack depth of six from an input system call (*e.g.*, sys_read()) should be microspeculated[2]
- LIST – functions specified in a profile (either generated automatically by STEM or manually specified) should be microspeculated

In order to support the COSAK [16] coverage policy, STEM maintains a `cosak_depth` variable via four operations: check, reset, increment, and decrement. Every time an input system call is encountered, the variable is reset to zero. The variable is checked during `ShouldSuperviseRoutine()` if the coverage policy is set to COSAK. The variable is incremented every time a new routine is entered during `STEM_Preamble()` and decremented during `STEM_Epilogue()`.

3.3 STEM Workflow

Although STEM can supervise an application from startup, STEM benefits from using PIN because PIN can attach to a running application. For example, if a network sensor detects anomalous data aimed at a web server, STEM can attach to the web server process to protect it while that data is being processed. In this way, applications can avoid the startup costs involved in instrumenting shared library loading, and can also avoid the overhead of the policy check for most normal input.

STEM starts by reading its configuration file, attaching some command and control functionality (described in Section 3.4), and then registering a callback to instrument each new function that it encounters. STEM's basic algorithm is distributed over the four main analysis routines. If STEM operates in profiling mode (see Section 6), then these analysis routines remain unused.

3.3.1 Memory Log

Since STEM needs to treat each function as a transaction, undoing the effects of a speculated transaction requires that STEM keep a log of changes made to memory during the transaction. The memory log is maintained by three functions: one that records the "old" memory value, one that inserts a marker into the memory log, and one that rolls back the memory log and optionally restores the "old" values. STEM inserts a call to `RecordPreMemWrite()` before an instruction that writes to memory. PIN determines the size of the write, so this analysis function can save the appropriate amount of data. Memory writes are only recorded for functions that should be supervised according to coverage policy. During `STEM_Preamble()`, PIN inserts a call to `InsertMemLogMarker()` to delimit a new function instance. This marker indicates that the last memory log maintenance function, `UnrollMemoryLog()`, should stop rollback after it encounters the marker. The rollback function deletes the entries in the memory log to make efficient use of the process's memory space. This function can also restore the "old" values stored in the memory log in preparation for repair.

3.3.2 STEM_Preamble()

This analysis routine performs basic record keeping. It increments the COSAK depth variable and maintains other statistics (number of routines supervised, *etc.*). Its most important tasks are to (1) check if supervision coverage policy should be reloaded and (2) insert a function name marker into the memory log if the current function should be supervised.

3.3.3 STEM_Epilogue()

STEM invokes this analysis routine immediately before a return instruction. Besides doing its part to maintain the COSAK depth variable, this analysis routine ensures that the application has a chance to self-heal before a transaction is completed. If the current function is being supervised, this routine interprets the application's repair policy (a form of integrity policy based on extensions to the Clark-Wilson integrity model, see Section 4 for details), invokes the repair procedure, and invokes the repair validation procedure. If both of these latter steps are successful or no repair is needed, then the transaction is considered to be successfully committed. If not, and an error *has* occurred, then STEM falls back to crashing the process (the current state of the art) by calling `abort()`.

This analysis routine delegates the setup of error virtualization to the repair procedure. The repair procedure takes the function name, current architectural context (*i.e.*, CPU register values), and a flag as input. The flag serves as an indication to the repair procedure to choose between normal cleanup or a "self-healing" cleanup. While normal cleanup always proceeds from `STEM_Epilogue()`, a self-healing cleanup can be invoked synchronously from `STEM_Epilogue()` or asynchronously from a signal handler. The latter case usually occurs when STEM employs a detector that causes a signal such as SIGSEGV to occur when it senses an attack.

Normal cleanup simply entails deleting the entries for the current function from the memory log. If self-healing is needed, then the values from the memory log are restored. In addition, a flag is set indicating that the process should undergo error virtualization, and the current function name is recorded.

3.3.4 SuperviseInstruction()

The job of this analysis routine is to intercept the instruction that immediately follows a RET instruction. By doing so, STEM allows the RET instruction to operate as it needs to on the architectural state (and by extension, the process stack). After RET has been invoked, if the flag for error virtualization is set, then STEM looks up the appropriate error virtualization value according to policy (either a vanilla EV value, or an EV value derived from the application's profile or repair policy). STEM then performs error virtualization by adjusting the value of the `%eax` register and resets the error virtualization flag. STEM ensures that the function returns appropriately by comparing the return address with the saved value of the instruction pointer immediately after the corresponding CALL instruction.

3.4 Additional Controls

STEM includes a variety of control functionality that assists the core analysis routines. The most important of these additional components intercepts signals to deal with dynamically loading configuration and selecting a suitable error virtualization value.

STEM defines three signal handlers and registers them with PIN. The first intercepts SIGUSR1 and sets a flag indicating that policy and configuration should be reloaded, although the actual reload takes place during the execution of the next `STEM_Preamble()`. The second signal handler intercepts SIGUSR2 and prints some runtime debugging information. The third intercepts SIGSEGV (for cases where detectors alert on memory errors, such as address space randomization). The handler then causes the repair procedure to be invoked, after it has optionally asked the user to select a response as detailed by the repair policy. Part of the response can include forwarding a snapshot of memory state to support automatically generating an exploit signature as done with the previous version of STEM for the FLIPS system [22].

STEM supports a variety of detection mechanisms, and it uses them to measure the integrity of the computation at various points in program execution and set a flag that indicates `STEM_Epilogue()` should initiate a self-healing response. Our current set of detectors includes one that detects an anomalous set of function calls (*i.e.*, a set of functions that deviate from a profile learned when STEM is in profiling mode) as well as a shadow stack that detects integrity violations of the return address or other stack frame information. STEM also intercepts a SIGSEGV produced by an underlying OS that employs address space randomization. We are currently implementing tainted dataflow analysis. This detector requires more extensive instrumentation, thereby limiting the supervision coverage policy to "ALL."

4 Repair Policy

Achieving a semantically correct response remains a key problem for self-healing systems. Executing through a fault or attack involves a certain amount of risk. Even if software could somehow ignore the attack itself, the best sequence of actions leading back to a safe state is an open question. The exploit may have caused a number of changes in state that corrupt execution integrity before an alert is issued. Attempts to self-heal must not only stop an exploit from succeeding or a fault from manifesting, but also repair execution integrity as much as possible. However, self-healing strategies that execute through a fault by effectively pretending it can be handled by the program code or other instrumentation may give rise to semantically incorrect responses. In effect, naive self-healing may provide a cure worse than the disease.

Figure 2 illustrates a specific example: an error may exist in a routine that determines the access control rights for a client. If this fault is exploited, a self-healing tech-

```
int login(UCRED creds)
{
  int authenticated = check_credentials(creds);
  if(authenticated) return login_continue();
  else return login_reject();
}
int check_credentials(UCRED credentials)
{
  strcpy(uname, credentials.username);
  return checkpassword(lookup(uname), credentials);
}
```

Figure 2: *Semantically Incorrect Response.* If an error arising from a vulnerability in `check_credentials` occurs, a self-healing mechanism may attempt to return a simulated error code from `check_credentials`. Any value other than `0` that gets stored in `authenticated` causes a successful login. What may have been a simple DoS vulnerability has been transformed into a valid login session by virtue of the "security" measures. STEM interprets *repair policy* to intelligently constrain return values and other application data.

nique like error virtualization may return a value that allows the authentication check to succeed. This situation occurs precisely because the recovery mechanism is oblivious to the semantics of the code it protects.

One solution to this problem relies on annotating the source code to (a) indicate which routines should not be "healed" or (b) to provide appropriate return values for such sensitive functions, but we find these techniques unappealing because of the need to modify source code. Since source-level annotations serve as a vestigial policy, we articulate a way to augment self-healing approaches with the notion of *repair policy*. A repair policy (or a recovery policy – we use the terms interchangeably) is specified separately from the source code and describes how execution integrity should be maintained after an attack is detected. Repair policy can provide a way for a user to customize an application's response to an intrusion attempt and can help achieve a completely automated recovery.

4.1 Integrity Repair Model

We provide a theoretical framework for repair policy by extending the Clark-Wilson Integrity Model (CW) [7] to include the concepts of (a) repair and (b) repair validation. CW is ideally suited to the problem of detecting when constraints on a system's behavior and information structures have been violated. The CW model defines rules that govern three major constructs: constrained data items (CDI), transformation procedures (TP), and integrity verification procedures (IVP). An information system is composed of a set of TPs that transition CDIs from one valid state to another. The system also includes IVPs that measure the integrity of the CDIs at various points of execution.

Although a TP should move the system from one valid state to the next, it may fail for a number of reasons (incorrect specification, a vulnerability, hardware faults, *etc.*). The purpose of an IVP is to detect and record this failure. CW does not address the task of returning the system to a valid state or formalize procedures that *restore* integrity. In contrast, repair policy focuses on ways to recover after an unauthorized modification. Our extensions supplements the CW model with primitives and rules for recovering from a policy violation and validating that the recovery was successful.

4.2 Interpreting Repair Policy

STEM interprets repair policy to provide a mechanism that can be selectively enforced and retrofitted to the protected application without modifying its source code (although *mapping* constraints to source-level objects assists in maintaining application semantics). As with most self-healing systems, we expect the repairs offered by this "behavior firewall" to be temporary constraints on program behavior — emergency fixes that await a more comprehensive patch from the vendor. One advantage of repair policy is that an administrator can "turn off" a broken repair policy without affecting the execution of the program — unlike a patch.

Repair policy is specified in a file external to the source code of the protected application and is used only by STEM (*i.e.*, the compiler, the linker, and the OS are not involved). This file describes the legal settings for variables in an aborted transaction. The basis of the policy is a list of relations between a transaction and the CDIs that need to be adjusted after error-virtualization, including the return address and return value. A complete repair policy is a wide-ranging topic; in this paper we consider a simple form that:

1. specifies appropriate error virtualization settings to avoid an incorrect return value that causes problems like the one illustrated in Figure 2

2. provides memory rollback for an aborted transaction

3. sets memory locations to particular values

Figure 3 shows a sample policy for our running example. The first statement defines a symbolic value. The latter three statements define an IVP, RP, and TP. The IVP defines a simple detector that utilizes STEM's shadow stack. The RP sets the return value to a semantically correct value and indicates that memory changes should be undone, and the TP definition links these measurement and repair activities together. An RP can contain a list of

```
symval AUTHENTICATION_FAILURE = 0;
ivp MeasureStack :=:
  ('raddress=='shadowstack[0]);
rp FixAuth :=:
  ('rvalue==AUTHENTICATION_FAILURE),
  (rollback);
tp check_credentials
  &MeasureStack :=: &FixAuth;
```

Figure 3: *Sample Repair Policy.* If the TP named `check_credentials` fails, then the memory changes made during this routine are reset and STEM stores the value `0` in the return value (and thus into `authenticated`), causing the login attempt to fail.

asserted conditions on CDIs that should be true after self-healing completes. The example illustrates the use of the special variable `'rvalue` (the apostrophe distinguishes it from any CDI named `rvalue`). This variable helps customize vanilla error virtualization to avoid problems similar to the one in Figure 2.

4.3 Limitations and Future Work

Our future work on STEM centers on improving the power and ease of use of repair policy. We intend to provide a mapping between memory layout and source-level variables. Cutting across layers of abstraction like this requires augmenting the mapping mechanism with a type system and the ability to handle variables that do not reside at fixed addresses. Second, while virtual proxies are a key aid to provide a semantically correct response, there is no explicit integration of virtual proxy behavior with repair policy specification. Third, we intend to explore the addition of formal logic to STEM so that it can reason about the constraints on the data involved in a transaction to learn the best response over time.

Finally, the information that a particular set of variables have been corrupted raises the possibility of notifying other hosts and application instances to proactively invoke repair procedures in order to protect against a widespread attack [21, 8, 35]. This sort of detection is helpful in creating a system that automatically tunes the security posture of an organization.

5 Virtual Proxies

Attempts to sandbox an application's execution must sooner or later allow the application to deal with global input and output sources and sinks that are beyond the control of the sandbox. Microspeculation becomes unsafe when the speculated process slice communicates with entities beyond the control of STEM. If a transaction is not idempotent (*i.e.*, it alters global state such as shared memory, network messages, *etc.*), then microspeculation must stop before that global state is changed. The self-healing system can no longer safely speculate a code slice: the results of execution up to that point must be committed, thus limiting microspeculation's effective scope.

Repair attempts may fall short in situations where an exploit on a machine (*e.g.*, an electronic funds transfer front-end) that is being "healed" has visible effects on another machine (*e.g.*, a database that clears the actual transfer). For example, if a browser exploit initiates a PayPal transaction, even though STEM can recover control on the local machine, the user will not have an automated recourse with the PayPal system.

Such situations require additional coordination between the two systems – microspeculation must span both machines. If both machines reside in the same administrative domain, achieving this *cooperative microspeculation* is somewhat easier, but we prefer a solution that works for situations like the PayPal example. While a self-healing system can record I/O, it cannot ask a communications partner to replay input or re-accept output. Doing so requires that the protocol (and potentially the network infrastructure) support speculative messaging and entails changing the partner's implementation so that it can rewind its own execution. Since STEM may not be widely deployed, we cannot rely on this type of explicit cooperation.

5.1 Solutions

We can achieve cooperative microspeculation in at least four ways, each of which expresses a tradeoff between semantic correctness and invasiveness.

1. **Protocol Modification** – Modify network or filesystem protocols and the network infrastructure to incorporate an explicit notion of speculation.

2. **Modify Communications Partner** – Modify the code of the remote entity so that it can cooperate when the protected application is microspeculating, and thus anticipate when it may be sending or receiving a "speculated" answer or request.

3. **Gradual Commits** – Transactions can be continuously limited in scope. All memory changes occurring *before* an I/O call are marked as not undoable. Should the microspeculated slice fail, STEM only undoes changes to memory made after the I/O call.

4. **Virtual Proxies** – Use buffers to record and replay I/O locally. Virtual proxies effectively serve as a man-in-the-middle during microspeculation to delay the effects of I/O on the external world.

While some network and application-level protocols may already include a notion of "replay" or speculative execution, implementing widespread changes to protocol specifications and the network infrastructure is fairly invasive. Nevertheless, it presents an interesting technical research challenge. Another interesting possibility is to modify the execution environment or code of the remote communications partner to accept notifications from a STEM-protected application. After receiving the notification, the remote entity speculates its own I/O. While this approach promises a sound solution, it violates our transparency requirements.

We choose to use a combination of virtual proxies and gradual commits because these solutions have the least impact on current application semantics and require a straightforward implementation. Since we are already "modifying" the local entity, we can avoid modifying the remote entity or any protocols. Using gradual commits and virtual proxies constrains the power of our solution, but we believe it is an acceptable tradeoff, especially as self-healing systems gain traction – they should perturb legacy setups as little as possible.

5.2 Design

I/O system calls that occur during the speculated portion of a process constitute a challenge for safely discarding speculated operations should an exploit occur. While speculation can immediately resume after an I/O call, the I/O call itself cannot be replayed or undone. If a fault or exploit occurs after the I/O call (but still in the microspeculated routine), then STEM cannot rewind to the beginning of the code slice. Rather, it can only unwind back to the I/O call. Memory and other state changes before the I/O call must remain in effect (we ignore for the moment explicit changes made as part of repair policy). This gradual process of commits is one way in which we can attempt to control uncertainty in the correctness of the response.

A virtual proxy serves as a delegate for a communications partner (*e.g.*, server, client, or peer) for the program that STEM is supervising. A virtual proxy is composed of a set of functions that modify a buffer that is bound during the scope of a supervised routine. The primary function of the virtual proxy is to allow STEM, as it speculates a slice of an application, to "take back" some output or "push back" some input. As a proof of concept, our current implementation only intercepts `read` and `write` calls. Virtual proxies are designed to handle this two-part problem.

Virtual Proxy Input In this case, an external component (such as a filesystem) is providing input. The code slice that contains this input call can either (a) successfully complete without an error or exploit, or (b) experience such a fault and have STEM attempt repair. In case (a), nothing need happen because STEM's state is consistent with the global state. In case (b), STEM must attempt a semantically correct repair – regardless of whether or not the input was legal or malformed/malicious. At this point, the external entity believes its state has changed (and therefore will not replay the input). In the optimal case, STEM should continue executing with what input *that was supposed to be consumed by the transaction removed from the input buffer*. Naturally, STEM cannot determine this on its own (and the speculated code slice is no help either – it evidently experienced a fault when processing this input). Instead, STEM can continue processing and draw from the virtual proxy's buffers during the next input request.

Virtual Proxy Output In order to deal with speculated output, STEM must buffer output until it requires input from the external component. At this point, STEM must allow the remote partner to make progress. This process of gradual commits is useful, but has the potential to delay too long and cause an application-level timeout. STEM does not currently deal with this issue. As with virtual proxy input, the speculated slice can (a) successfully complete without an error or exploit, or (b) experience such a fault and have STEM attempt a repair. In case (a), gradual commits suffice, as the output calls simply finish. In case (b), the external component has been given a message it should not have. If the virtual proxy were not operating, a STEM-supervised application would need to ask for that output to be ignored. The virtual proxy allows STEM to buffer output until the microspeculated slice successfully completes. If the slice fails, then STEM instructs the virtual proxy to discard the output (or replace it).

5.3 Limitations and Future Work

Although virtual proxies help address the external I/O problem for microspeculation, they are not a perfect solution. In the case where STEM is supervising the processing of input, the virtual proxy can only buffer a limited amount of input – and it is not clear how to selectively discard portions of that input should a transaction fail. In the cases where STEM supervises the sending of output, the virtual proxy buffers the output until STEM requests input from the remote communications partner. At this point, STEM has reached the edge of our ability to safely microspeculate, and without further support in the virtual proxy that explicitly communicates with the remote partner, STEM must stop speculating and finally give the data to the remote partner.

One interesting problem is to use multiple virtual proxies to classify and identify multiple conversation streams. This information is not present at the level of read and write system calls, and STEM would need to break through layers of abstraction to support this ability. Finally, since the virtual proxy is under STEM's control, STEM can attempt to construct a memory and behavior model of the remote communications partner to determine if it is behaving in a malicious fashion.

6 Behavior Models

Although STEM uses a number of detection strategies (including a shadow stack), STEM also provides for host-based anomaly detection. This type of detection helps identify previously unknown vulnerabilities and exploits, but depends on the system having a model or profile of normal behavior. STEM collects aspects of data and control flow to learn an application's behavior profile. STEM can leverage the information in the profile to detect misbehavior (*i.e.,* deviation from the profile) and automatically validate repairs to ensure that self-healing achieves normal application behavior.

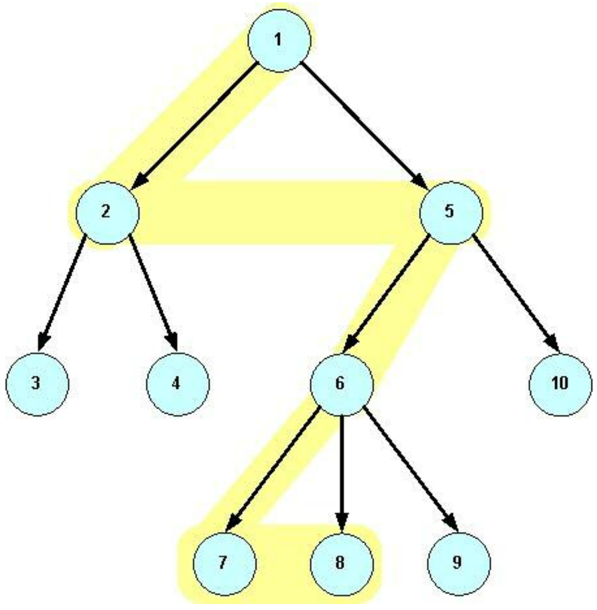

Figure 4: *Example of Computing Execution Window Context*. Starting from function 8, we traverse the graph beginning from the previously executed siblings up to the parent. We recursively repeat this algorithm for the parent until we either reach the window width or the root. In this example, the window contains functions 7, 6, 5, 2, 1. Systems that examine the call stack would only consider 6, 5, and 1 at this point.

In profiling mode, STEM dynamically analyzes all function calls made by the process, including regular functions and library calls as well as system calls. Previous work typically examines only system calls or is driven by static analysis. STEM collects a feature set that includes a mixture of parent functions and previous sibling functions. STEM generates a record of the observed return values for various invocations of each function.

A behavior profile is a graph of execution history records. Each record contains four data items: an identifier, a return value, a set of argument values, and a context. Each function name serves as an identifier (although address/callsites can also be used). A mixture of parents and previous siblings compose the context. The argument and return values correspond to the argument values at the time that function instance begins and ends, respectively. STEM uses a pair of analysis functions (inserted at the start and end of each routine) to collect the argument values, the function name, the return value, and the function context.

Each record in the profile helps to identify an instance of a function. The feature set "unflattens" the function namespace of an application. For example, `printf()` appears many times with many different contexts and return values, making it hard to characterize. Considering every occurrence of `printf()` to be the same instance reduces our ability to make predictions about its behavior. On the other hand, considering all occurrences of `printf()` to be separate instances combinatorially increases the space of possible behaviors and similarly reduces our ability to make predictions about its behavior in a reasonable amount of time. Therefore, we need to construct an "execution context" for each function based on both control (predecessor function calls) and data (return & argument values) flow. This context helps collapse *occurrences* of a function into an *instance* of a function. Figure 4 shows an example context window.

During training, one behavior aspect that STEM learns is which return values to predict based on execution contexts of varying window sizes. The general procedure attempts to compute the prediction score by iteratively increasing the window size and seeing if additional information is revealed by considering the extra context.

We define the return value "predictability score" as a value from zero to one. For each context window, we calculate the "individual score": the relative frequency of this particular window when compared with the rest of the windows leading to a function. The predictability score for a function F is the sum of the individual scores that lead to a single return value. Figure 5 displays an example of this procedure. We do not consider windows that contain smaller windows leading to a single return value since the information that they impart is already subsumed by the smaller execution context. For example, in Figure 5, we do not consider all windows with a suffix of AF (*i.e.,* $*AF$).

Limitations STEM relies on PIN to reliably detect

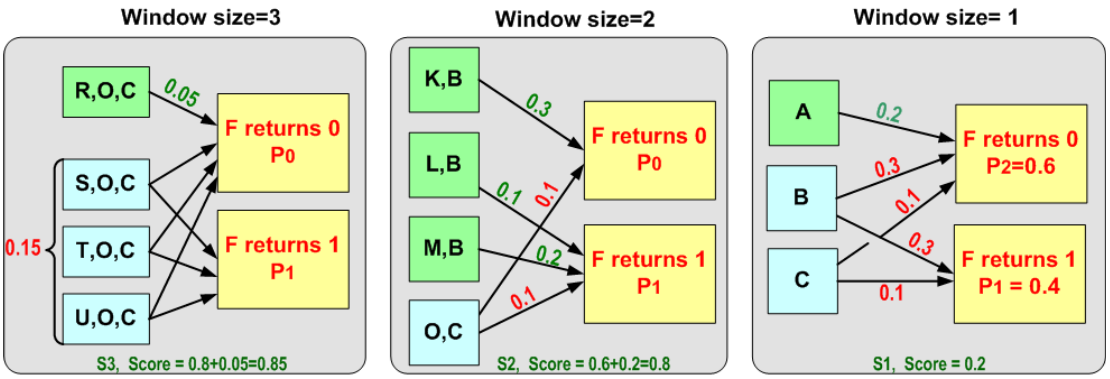

Figure 5: *Example of Computing Return Value Predictability (predictability score)*. The figure illustrates the procedure for function F and for two return values 0 & 1 for three window sizes. The arrow labels indicate what percentage of instances for the given window will lead to the return value of F when compared with the rest of the windows. For window size 1 (S1) we have three predicate functions (A, B, and C) with only one, A, leading to a unique return value with score 0.2. This score is the relative frequency of window AF,[2] when compared with all other windows leading to F, for all return values. We add a score to the total score when a window leads to single return value of F since this situation is the only case that "predicts" a return value. We consider only the smallest windows that lead to a single value (*e.g.,* A is no longer considered for S2 and KB, LB, MB for S3) because larger windows do not add anything to our knowledge for the return value.

returns from a function. Detecting function exit is difficult in the presence of optimizations like tail recursion. Also, since the generated profile is highly binary-dependent, STEM should recognize when an older profile is no longer applicable (and a new one needs to be built), *e.g.,* as a result of a new version of the application being rolled out, or due to the application of a patch.

7 Evaluation

The goal of our evaluation is to characterize STEM's impact on the normal performance of an application. STEM incurs a relatively low performance impact for real-world software applications, including both interactive desktop software as well as server programs. Although the time it takes to self-heal is also of interest, our experiments on synthetic vulnerabilities show that this amount of time depends on the complexity of the repair policy (*i.e.*, how many memory locations need to be adjusted) and the memory log rollback. Even though memory log rollback is an $O(n)$ operation (we discuss a possible optimization below), STEM's self-healing and repair procedure usually takes under a second (using the `x86 rdtsc` instruction we observe an average of 15 milliseconds) to interpret the repair policy for these vulnerabilities.

Table 1: *Impact on Apache Excluding Startup*. We tested STEM's impact on two versions of Apache by starting Apache in single-threaded mode (to force all requests to be serviced sequentially by the same thread). We then attach STEM after verifying that Apache has started by viewing the default homepage. We use `wget` to recursively retrieve the pages of the online manual included with Apache. The total downloaded material is roughly 72 MB in about 4100 files. STEM causes a 74.85% slowdown, far less than the tens of thousands factor when including startup. Native execution of Apache 2.0.53 takes 0.0626 seconds per request; execution of the same under STEM takes 0.1095 seconds per request. For a newer version of Apache (2.2.4), we observe a slight improvement to 72.54%.

Apache	Native (s)	STEM (s)	Impact %
`v2.0.53`	3746	6550	74.85%
`v2.2.4`	16215	27978	72.54%

Of more general concern is whether or not STEM slows an application down to the point where it becomes apparent to the end-user. Even though STEM has a rather significant impact on an application's startup time (as shown in Table 2), STEM does not have a human-discernible impact when applied to regular application operations. For example, Firefox remains usable for casual web surfing when operating with STEM. In addition, playing a music file with `aplay` also shows no sign of sound degradation – the only noticeable impact comes during startup. Disregarding this extra time, the difference between `aplay`'s native performance and its performance under STEM is about 2 seconds. If STEM is attached to `aplay` after the file starts playing, there is an eight second delay followed by playback that proceeds with only a 3.9% slowdown. Most of the performance penalty shown in Table 2 and Table 3 is exaggerated by

Table 2: *Performance Impact Data*. Attaching STEM at startup to dynamically linked applications incurs a significant performance penalty that lengthens the total application startup time. This table lists a variety of programs where the time to execute is dominated by the increased startup time. Although most applications suffer a hefty performance hit, the majority of the penalty occurs during application startup and exit. Note that `aplay` shows fairly good performance; a roughly six-minute song plays in STEM for 88 seconds longer than it should – with 86 of those seconds coming during startup, when the file is not actually being played.

Application	**Native (s)**	**STEM (s)**	**Slowdown**
`aplay`	371.0125	459.759	0.239
`arch`	0.001463	14.137	9662.021
`xterm`	0.304	215.643	708.352
`echo`	0.002423	17.633	7276.342
`false`	0.001563	16.371	10473.088
`Firefox`	2.53725	70.140	26.644
`gzip-h`	4.51	479.202	105.253
`gzip-k`	0.429	58.954	136.422
`gzip-d`	2.281	111.429	47.851
`md5-k`	0.0117	32.451	2772.589
`md5-d`	0.0345	54.125	1567.841
`md5-h`	0.0478	70.883	1481.908
`ps`	0.0237	44.829	1890.519
`true`	0.001552	16.025	10324.387
`uname`	0.001916	19.697	10279.271
`uptime`	0.002830	27.262	9632.215
`date`	0.001749	26.47	15133.362
`id`	0.002313	24.008	10378.592

the simple nature of the applications. Longer-running applications experience a much smaller impact relative to total execution, as seen by the gzip, md5sum, and Firefox results.

Most of the work done during startup loads and resolves libraries for dynamically linked applications. STEM can avoid instrumenting this work (and thus noticeably reduce startup time) in at least two ways. The first is to simply not make the application dynamically linked. We observed for some small test applications (including a program that incorporates the example shown in Figure 2 from Section 4) that compiling them as static binaries reduces execution time from fifteen seconds to about five seconds. Second, since PIN can attach to applications after they have started (in much the same way that a debugger does), we can wait until this work completes and then attach STEM to protect the mainline code execution paths. We used this capability to attach STEM to Firefox and Apache after they finish loading (we measured the performance impact on Apache using this method; see Table 1). Also, as mentioned in Section 3, we can allow the application to begin executing normally and only attach STEM when a network anomaly detector issues an IDS alert. Finally, it may be acceptable for certain long-running applications (*e.g.*, web, mail, database, and DNS servers) to amortize this long startup time (on the order of minutes) over the total execution time (on the order of weeks or months).

Table 3: *Performance Without (Some) Startup*. We remove a well-defined portion of the application's initialization from the performance consideration in Table 2. Removing supervision of this portion of the startup code improves performance over full supervision. The remaining run time is due to a varying amount of startup code, the application itself, and cleanup/exit code. In order to completely eliminate application startup from consideration, we attach to Apache after its initialization has completed. We present those results in Table 1.

Application	**STEM-init (s)**	**Revised Slowdown**
`arch`	3.137	2143.22
`xterm`	194.643	639.273
`echo`	5.633	2323.803
`false`	4.371	2795.545
`Firefox`	56.14	21.128
`gzip-h`	468.202	102.814
`gzip-k`	47.954	110.780
`gzip-d`	100.429	43.025
`md5-k`	20.451	1746.948
`md5-d`	42.125	1220.014
`md5-h`	58.883	1230.862
`ps`	31.829	1341.996
`true`	5.025	3236.758
`uname`	8.697	4538.144
`uptime`	15.262	5391.932
`date`	14.47	8272.299
`id`	13.008	5622.865

7.1 Experimental Setup

We used multiple runs of applications that are representative of the software that exists on current Unix desktop environments. We tested aplay, Firefox, gzip, md5sum, and xterm, along with a number of smaller utilities: arch, date, echo, false, true, ps, uname, uptime, and id. The applications were run on a Pentium M 1.7 GHz machine with 2 GB of memory running Fedora Core 3 Linux. We used a six minute and ten second WAV file to test `aplay`. To test both `md5sum` and `gzip`, we used three files: *httpd-2.0.53.tar.gz*, a Fedora Core kernel (*vmlinuz-2.6.10-1.770_FC3smp*), and the */usr/share/dict/linux.words* dictionary. Our Firefox instance simply opened a blank page. Our xterm test creates an xterm and executes the `exit` command. We also tested two versions of `httpd` (2.0.53 and 2.2.4) by attaching STEM after Apache starts and using `wget` to recursively download the included manual from another machine on the same network switch. Doing so gives us a way to measure STEM's impact on normal performance excluding startup (shown in Table 1). In the tables, the suffixes for gzip and md5sum indicate the kernel image (k), the httpd tarball (h), and the dictionary (d).

Memory Log Enhancements We can improve performance of supervised routines by modifying the memory log implementation (currently based on a linked list). One way to improve performance is to preallocate memory slots based on the typical memory use of each supervised function. If we can bound the number of stores in a piece of code (*e.g.*, because STEM or another profiling tool has observed its execution), then STEM can preallocate an appropriately sized buffer.

8 Conclusion

Self-healing systems face a number of challenges before they can be applied to legacy applications and COTS software. Our efforts to improve STEM focus on four specific problems: (1) applying STEM's microspeculation and error virtualization capabilities in situations where source code is unavailable, (2) helping create a behavior profile for detection and repair, (3) improving the correctness of the response by providing a mechanism to interpret *repair policy*, and (4) implementing *virtual proxies* to help deal with speculated I/O. These solutions collectively provide a more streamlined version of STEM that represents a significant improvement in both features and performance: our current implementation imposes a 74% impact for whole-application supervision (versus the previous 30% impact for a single supervised routine and a 3000X slowdown for whole-application supervision).

Acknowledgments

We deeply appreciate the insightful and constructive comments made by the anonymous reviewers. This material is based on research sponsored by the Air Force Research Laboratory under agreement number FA8750-06-2-0221, and by the National Science Foundation under NSF grants CNS-06-27473, CNS-04-26623 and CCF-05-41093. We authorize the U.S. Government to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-Flow Integrity: Principles, Implementations, and Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2005).

[2] BARATLOO, A., SINGH, N., AND TSAI, T. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the USENIX Annual Technical Conference* (June 2000).

[3] BHATKAR, S., CHATURVEDI, A., AND SEKAR., R. Improving Attack Detection in Host-Based IDS by Learning Properties of System Call Arguments. In *Proceedings of the IEEE Symposium on Security and Privacy* (2006).

[4] BROWN, A., AND PATTERSON, D. A. Rewind, Repair, Replay: Three R's to dependability. In 10^{th} *ACM SIGOPS European Workshop* (Saint-Emilion, France, Sept. 2002).

[5] CANDEA, G., AND FOX, A. Crash-Only Software. In *Proceedings of the* 9^{th} *Workshop on Hot Topics in Operating Systems (HOTOS-IX)* (May 2003).

[6] CHARI, S. N., AND CHENG, P.-C. BlueBoX: A Policy–driven, Host–Based Intrusion Detection System. In *Proceedings of the* 9^{th} *Symposium on Network and Distributed Systems Security (NDSS 2002)* (2002).

[7] CLARK, D. D., AND WILSON, D. R. A Comparison of Commercial and Military Computer Security Policies. In *Proceedings of the IEEE Symposium on Security and Privacy* (1987).

[8] COSTA, M., CROWCROFT, J., CASTRO, M., AND ROWSTRON, A. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP)* (2005).

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

[10] CUI, W., PEINADO, M., WANG, H. J., AND LOCASTO, M. E. ShieldGen: Automated Data Patch Generation for Unknown Vulnerabilities with Informed Probing. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2007).

[11] DEMSKY, B., AND RINARD, M. C. Automatic Detection and Repair of Errors in Data Structures. In *Proceedings of the* 18^{th} *Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications* (October 2003).

[12] FENG, H. H., KOLESNIKOV, O., FOGLA, P., LEE, W., AND GONG, W. Anomaly Detection Using Call Stack Information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy* (May 2003).

[13] GAO, D., REITER, M. K., AND SONG, D. Gray-Box Extraction of Execution Graphs for Anomaly Detection. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2004).

[14] GIFFIN, J. T., DAGON, D., JHA, S., LEE, W., AND MILLER, B. P. Environment-Sensitive Intrusion Detection. In *Proceedings of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)* (September 2005).

[15] HOFMEYR, S. A., SOMAYAJI, A., AND FORREST, S. Intrusion Detection System Using Sequences of System Calls. *Journal of Computer Security 6*, 3 (1998), 151–180.

[16] HTTP://SERG.CS.DREXEL.EDU/COSAK/INDEX.SHTML.

[17] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. Secure Execution Via Program Shepherding. In *Proceedings of the* 11^{th} *USENIX Security Symposium* (August 2002).

[18] LAM, L. C., AND CKER CHIUEH, T. Automatic Extraction of Accurate Application-Specific Sandboxing Policy. In *Proceedings of the* 7^{th} *International Symposium on Recent Advances in Intrusion Detection* (September 2004).

[19] LIANG, Z., AND SEKAR, R. Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers. In *Proceedings of the* 12^{th} *ACM Conference on Computer and Communications Security (CCS)* (November 2005).

[20] LOCASTO, M. E., CRETU, G. F., STAVROU, A., AND KEROMYTIS, A. D. A Model for Automatically Repairing Execution Integrity. Tech. Rep. CUCS-005-07, Columbia University, January 2007.

[21] LOCASTO, M. E., SIDIROGLOU, S., AND KEROMYTIS, A. D. Software Self-Healing Using Collaborative Application Communities. In *Proceedings of the* 13^{th} *Symposium on Network and Distributed System Security (NDSS 2006)* (February 2006), pp. 95–106.

[22] LOCASTO, M. E., WANG, K., KEROMYTIS, A. D., AND STOLFO, S. J. FLIPS: Hybrid Adaptive Intrusion Prevention. In *Proceedings of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)* (September 2005), pp. 82–101.

[23] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD., K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of Programming Language Design and Implementation (PLDI)* (June 2005).

[24] NEWSOME, J., BRUMLEY, D., AND SONG, D. Vulnerability–Specific Execution Filtering for Exploit Prevention on Commodity Software. In *Proceedings of the* 13^{th} *Symposium on Network and Distributed System Security (NDSS 2006)* (February 2006).

[25] NEWSOME, J., AND SONG, D. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the* 12^{th} *Symposium on Network and Distributed System Security (NDSS)* (February 2005).

[26] OPLINGER, J., AND LAM, M. S. Enhancing Software Reliability with Speculative Threads. In *Proceedings of the* 10^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)* (October 2002).

[27] PROVOS, N. Improving Host Security with System Call Policies. In *Proceedings of the* 12^{th} *USENIX Security Symposium* (August 2003), pp. 207–225.

[28] QIN, F., TUCEK, J., SUNDARESAN, J., AND ZHOU, Y. Rx: Treating Bugs as Allergies – A Safe Method to Survive Software Failures. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP)* (2005).

[29] RINARD, M., CADAR, C., DUMITRAN, D., ROY, D., LEU, T., AND W BEEBEE, J. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings* 6^{th} *Symposium on Operating Systems Design and Implementation (OSDI)* (December 2004).

[30] SIDIROGLOU, S., LAADAN, O., KEROMYTIS, A. D., AND NIEH, J. Using Rescue Points to Navigate Software Recovery (Short Paper). In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2007).

[31] SIDIROGLOU, S., LOCASTO, M. E., BOYD, S. W., AND KEROMYTIS, A. D. Building a Reactive Immune System for Software Services. In *Proceedings of the USENIX Annual Technical Conference* (April 2005), pp. 149–161.

[32] SMIRNOV, A., AND CHIUEH, T. DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks. In *Proceedings of the* 12^{th} *Symposium on Network and Distributed System Security (NDSS)* (February 2005).

[33] SOMAYAJI, A., AND FORREST, S. Automated Response Using System-Call Delays. In *Proceedings of the* 9^{th} *USENIX Security Symposium* (August 2000).

[34] SUH, G. E., LEE, J. W., ZHANG, D., AND DEVADAS, S. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the* 11^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)* (October 2004).

[35] TUCEK, J., LU, S., HUANG, C., XANTHOS, S., ZHOU, Y., NEWSOME, J., BRUMLEY, D., AND SONG, D. Sweeper: A Lightweight End-to-End System for Defending Against Fast Worms. In *EuroSys* (2007).

[36] XU, J., NING, P., KIL, C., ZHAI, Y., AND BOOKHOLT, C. Automatic Diagnosis and Response to Memory Corruption Vulnerabilities. In *Proceedings of the* 12^{th} *ACM Conference on Computer and Communications Security (CCS)* (November 2005).

Notes

[1] This limitation is especially relevant for financial and scientific applications, where a function's return value is more likely to be incorporated into the mainline calculation.

[2] Part of the CoSAK, or Code Security Analysis Kit, study found that most vulnerabilities in a set of popular open source software occur within six function calls of an input system call. If one considers a layer or two of application-internal processing and the existing (but seldom thought of from an application developer's standpoint) multiple layers within C library functions, this number makes sense.