Resilient Authenticated Execution of Critical Applications in Untrusted Environments

Michael S. Kirkpatrick, Gabriel Ghinita, *Member*, *IEEE*, and Elisa Bertino, *Fellow*, *IEEE*

Abstract—Modern computer systems are built on a foundation of software components from a variety of vendors. While critical applications may undergo extensive testing and evaluation procedures, the heterogeneity of software sources threatens the integrity of the execution environment for these trusted programs. For instance, if an attacker can combine an application exploit with a privilege escalation vulnerability, the operating system (OS) can become corrupted. Alternatively, a malicious or faulty device driver running with kernel privileges could threaten the application. While the importance of ensuring application integrity has been studied in prior work, proposed solutions immediately terminate the application once corruption is detected. Although, this approach is sufficient for some cases, it is undesirable for many critical applications. In order to overcome this shortcoming, we have explored techniques for leveraging a trusted virtual machine monitor (VMM) to observe the application and potentially *repair damage that occurs*. In this paper, we describe our system design, which leverages efficient coding and authentication schemes, and we present the details of our prototype implementation to quantify the overhead of our approach. Our work shows that it is feasible to build a resilient execution environment, even in the presence of a corrupted OS kernel, with a reasonable amount of storage and performance overhead.

Index Terms—Operating systems, security, virtual machine monitors.

1 Introduction

Several classes of applications, such as military, health or infrastructure monitoring software, are highly critical; compromising their correct execution may have dire consequences. Typically, such applications originate at trustworthy sources, and they undergo a thorough testing process, possibly including formal verification. Consequently, it is reasonable to consider these programs to be trusted and free of exploitable vulnerabilities. To ensure that the code executes correctly, it is vital to provide strong guarantees that the critical application is isolated from untrusted code. However, modern computing practices tend to make such isolation impossible.

Specifically, trusted software frequently runs on top of a commercial off-the-shelf (COTS) operating system (OS). Such COTS systems tend to be very complex and proprietary, preventing a rigorous security evaluation and formal analysis. Furthermore, the presence of legacy applications on the same machine may make secure alternatives infeasible. Proper isolation of the trusted processes, then, relies on the integrity and correctness of the OS. However, security vulnerabilities in the untrusted applications and/or the COTS OS destroy the guarantees of isolation and may lead to compromise of the trusted code.

- *M.S. Kirkpatrick is with the Department of Computer Science, James Madison University, 701 Carrier Drive, MSC 4103, Harrisonburg, VA 22807. E-mail: kirkpams@jmu.edu.*
- *G. Ghinita is with the Department of Computer Science, University of Massachusetts, 100 William T. Morrissey Boulevard, Boston, MA 02125. E-mail: gabriel.ghinita@umb.edu.*
- *E. Bertino is with the Department of Computer Science, LWSN Building, Purdue University, West Lafayette, IN 47907. E-mail: bertino@cs.purdue.edu.*

Manuscript received 7 Apr. 2011; revised 5 Jan. 2012; accepted 15 Feb. 2012; published online 29 Feb. 2012.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSCSI-2011-04-0068.
Digital Object Identifier no. 10.1109/TDSC.2012.25.

Memory corruption attacks are among the most frequently occurring security violations, accounting for 70 percent of the total number of CERT advisories between 2000 and 2003 [1]. A common approach is for an attacker to provide malformed input to an application to change its execution. Such attacks can be especially devastating if the target is the OS itself. As today's attacks predominantly employ these tactics, the security literature has primarily focused on protecting an application from external sources and/or protecting the OS from compromise. However, recent high-profile attacks suggest that future adversaries may be very sophisticated, well funded, and state-backed, and may have very precise targets. For instance, Stuxnet combined multiple exploits in order to disrupt the proper execution of a particular programmable logic controller (PLC) [2]; the worm was harmless to machines that did not have this PLC installed.

To reflect the changing nature of threats, our work assumes a very powerful adversarial model, in which we assume that the OS has already been compromised. That is, the attacker has already "won the game," according to traditional security threat models. However, we assume that the adversary's goal is to leverage the OS privileges in order to modify the memory image of the critical application. For instance, if the critical application computes missile trajectories for a military operation, the adversary's goal may be simply to skew the results. Our goal, then, is twofold. First, we aim to identify such corruption whenever it occurs. Second, if the damage is reasonably small, we desire to *repair the memory image dynamically*, allowing the process to continue normal execution.

Application recovery is a complex task and may take place at different levels. Recovery-oriented computing (ROC) [3] involves designing rapid failure recovery mechanisms into new and existing applications. The resulting programs include, the ability to "undo" errors by returning to a good state. The approach that we adopt in this paper is to use a trusted virtual machine monitor (VMM) to detect and repair the corrupted application memory pages. As such, our work can be seen as a technique for transparently incorporating ROC into the execution environment without modifying the original application code.

Existing research has already acknowledged the importance of protecting critical applications against an untrusted and/or compromised execution environment. Like ours, a common approach is to employ a trusted VMM that mediates interactions between the OS and the critical application, and restricts the kernel's access with respect to the memory space of the protected application. These VMM-based solutions generally fall into two broad categories: *memory authentication* and *memory duplication* approaches.

- *Memory authentication.* In this category, a trusted component (e.g., the VMM) applies cryptographic techniques to validate the integrity of the application's memory image during execution. For instance, Terra [4] provides trusted applications with isolated virtual machines, and uses cryptographic hashes of the software image to allow remote attestation to a third party. The hashes, along with a summary of the hardware and software layers running on the virtual machine, are signed with a private key stored in tamper-resistant hardware. Overshadow [5] provides both confidentiality and integrity of critical applications by encrypting the memory image. As the program executes, the VMM authenticates and decrypts pages as they are referenced.
- *Memory duplication.* NICKLE [6] protects the OS from rootkits by securely storing a cloned image of the kernel. At boot time, the trusted VMM creates a copy of the kernel in a portion of memory that is inaccessible to the guest OS. As the system runs, only instructions retrieved from this copy are permitted to execute in kernel mode. That is, if a rootkit has been installed after the system boots, it *cannot* execute, as its instructions do not exist in the protected space. While NICKLE protects the OS at runtime, it cannot protect critical applications if the original kernel image is malicious.

Observe that these approaches primarily focus on *detection* of memory corruption. Memory authentication mechanisms terminate the protected application if corruption is detected. Similarly, memory duplication techniques detect and prevent the introduction of malicious *kernel* code, but do not protect application code. Furthermore, for critical applications, detection of corruption alone is inadequate, as disruption of the execution may have disastrous consequences. That is, the attacker still wins, even if the corrupted process is terminated.

To ensure critical application recovery, memory authentication must be combined with mechanisms for application checkpointing [7]. Such mechanisms allow the critical application to be reinstated to a valid configuration saved previously. Fig. 1a illustrates alternatives for checkpointing: the memory image can be saved to protected local storage, or to a remote server if there are concerns that even the local

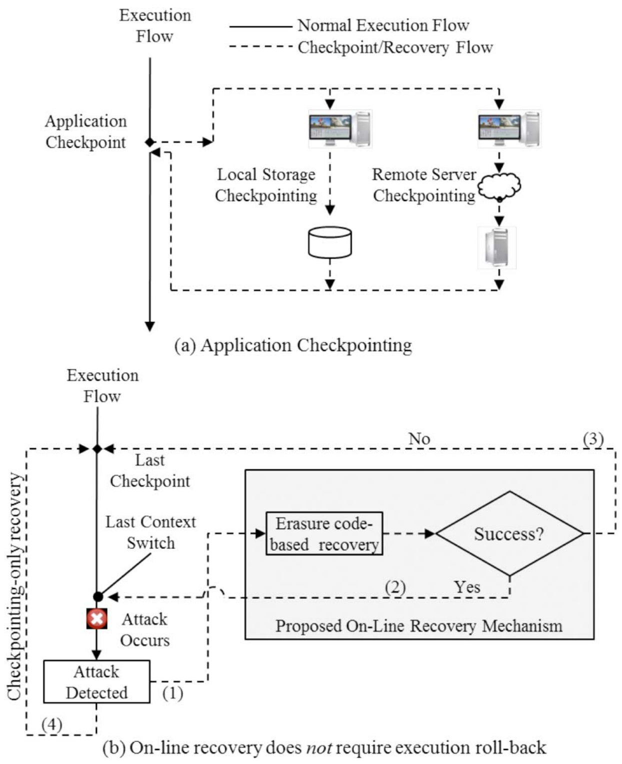
Execution Flow
Normal Execution Flow
Checkpoint/Recovery Flow
Application Checkpoint
Local Storage Checkpointing
Remote Server Checkpointing
(a) Application Checkpointing
Execution Flow
No
(3)
Last Checkpoint
Last Context Switch
Erasure code-based recovery
Success?
(2)
Yes
Attack Occurs
Proposed On-Line Recovery Mechanism
Attack Detected
(1)
(4)
Checkpointing-only recovery
(b) On-line recovery does *not* require execution roll-back

Fig. 1. Overview of proposed approach for online critical application recovery.

storage may be compromised. However, application-level checkpointing has several drawbacks. First, it is not secure. Since checkpointing and recovery are performed at a higher layer than the OS, there is no protection against a malicious or compromised operating system. Therefore, the integrity of the critical application is compromised. Second, the *granularity* of recovery is coarse, as correct execution can only be resumed from the previously saved application-level checkpoint. However, between the time the last checkpoint was created and the attack is staged, the critical application performed more computation. The results of such computation are lost in the recovery process. Third, saving the memory image to an external/remote system incurs considerable overhead, therefore can only be done with limited frequency, otherwise most of the CPU time will have to be spent on checkpointing, rather than performing useful computation.[1] This limits the frequency of checkpointing, which in turn exacerbates the second drawback: namely, the lower the frequency, the more significant the portions of the critical application execution that must be rolled-back.

We address these limitations through a novel framework for resilient execution of critical applications running in untrusted environments. Specifically, we propose and evaluate a VMM-based *online recovery mechanism* that is secure and achieves finer granularity of recovery compared to application-level checkpointing. To meet these goals, we rely on a combination of memory image authentication and duplication. Fig. 1b illustrates the functionality of the

1. For instance, the overhead incurred by network latency and transmission time to a remote server is already considerable, not accounting for the time required to access the memory data structures.

proposed approach: when an attack is detected, an online memory image reconstruction procedure is initiated (arrow 1 in the diagram). This procedure aims to restore the correct image based on a small amount of redundant information which is kept in protected storage, outside the reach of the OS or other untrusted software components. If reconstruction succeeds, (arrow 2) then the application will continue with the image that existed the last time a context switch was performed for the protected application (i.e., the last context switch before the attack). If online reconstruction fails (arrow 3), then the system reverts back to the last saved application-level checkpoint, which is equivalent to what existing checkpointing techniques do (arrow 4). This results in a considerable amount of execution being rolled back, namely everything that was executed between the last checkpoint and the last context switch. Clearly, the recovery granularity of our proposed method for online recovery is considerably more fine grained than checkpointing. As we show experimentally in Section 5, the success rate of online reconstruction is high under reasonable corruption attacks. Therefore, in most cases we avoid the rolling back of execution, resulting in a practical and efficient framework for achieving critical application availability.

For the memory duplication portion of our scheme, we have evaluated three approaches. First, the naïve approach simply clones the application memory image in its entirety. While efficient in terms of speed, the space requirements of this approach may be prohibitive. Second, we employ error correcting *digital fountain codes (DFCs)* [8], which have been proposed for reliable communication over unreliable network channels. We focus on *LT* codes, a DFC instantiation, which take a message consisting of K blocks and encode the data into $N > K$ blocks. LT codes provide a probabilistic guarantee of reconstructing the original message if a small number of the N blocks are corrupted. Our LT approach is efficient in space, requiring less than 25 percent extra storage, but reconstruction must be done at every page reference. Our third approach, which provides a balance between the first two, is to use Reed-Solomon (RS) codes [9], which append an array of data with a small number of parity bits for error correction. While Reed-Solomon encoding and decoding is significantly slower than the same procedures for LT codes, these procedures must only be executed when corruption occurs, producing better aggregate performance.

The rest of the paper is organized as follows: we start with background information on error correcting codes in Section 2. Section 3 describes the basic structure of the proposed architecture, whereas Section 4 provides technical details. We present the results of our experimental evaluation in Section 5. Section 6 compares our solution with related work, and we conclude in Section 7.

2 Error Correcting Codes

Our system employs error correcting codes to recover corrupted memory images. We consider two codes: Reed-Solomon [9] which is a parity-checksum based code, as well as LT codes [10], a digital fountain rateless code.

RS codes rely on interpolation of polynomials in a finite field, and for each block of k symbols of input generates $n > k$ symbols of output. The resulting $RS(n, k)$ code can correct up to t symbol errors, where $t = (n - k)/2$. In our setting, each symbol is a byte of data. A common setting is $RS(255, 223)$ which can correct 16 bytes for each block of 223 bytes using a 32-bytes checksum. The checksum of a message, also referred to as a *syndrome*, is obtained by multiplying the k-byte message, interpreted as the coefficients of a k-degree polynomial, with $x^{(n-k)}$ and computing the remainder modulo a generator polynomial g of degree $2t$. Decoding is equivalent to computing the roots of a n-degree polynomial. The RS code is rather compute-intensive, but has the advantage that in absence of errors, the original message can be accessed directly, as opposed to fountain codes that are discussed next.

Digital fountain codes have been designed for error correction in packet-switched communication networks, such as the Internet. In packet switched networks, data losses typically occur at the packet level, i.e., a packet is either correctly received, or it is entirely dropped (forwarding routers may decide to drop a packet along the transmission path if certain error-detection checksums fail). In this setting, having self-contained redundant information within each packet may not be an effective solution. The idea behind DFC is to reencode the message to be sent, such that redundant error-correcting information is shared by multiple packets. Specifically, given K payload packets, DFC provide mechanisms that may generate a potentially infinite sequence of packets, and any $N > K$ such packets (where N is only slightly larger than K) are sufficient to reconstruct the original message. Each transformed message is obtained by performing an exclusive-or operation on a subset of the original K packets. Due to their property that an infinite number of packets may be generated, such codes are called *rateless*. The term "fountain" is an analogy that captures the fact that the receiver needs to collect a number of *any* N packets to recover the message, similarly to collecting drops of water from under a fountain. We present the details of LT codes operation in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TDSC.2012.25. The advantage of LT codes over RS codes is the faster encoding and decoding time. However, decoding has to be performed even if there are no errors, since the original message (i.e., memory block) is not available in direct form.

3 System Overview

Our system design consists of using a VMM to incorporate error-correcting codes into the x86 memory architecture. The goal of our design is to protect the code and data for applications executing within an untrusted OS environment. That is, we aim to provide a robust defensive mechanism that protects the application even if the OS kernel has been corrupted. As described in Section 6, existing work in this area that has been designed to protect application data has provided only a detect-and-terminate approach. Our goal is more ambitious, as we want to provide a probabilistic guarantee that the application can recover from memory corruption and continue processing.

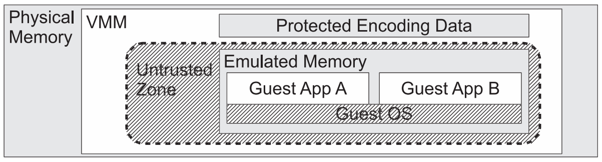

Fig. 2. VMM-based system architecture.

3.1 Architecture and Approach

Fig. 2 shows a high-level view of our system. The guest OS executes within an *untrusted zone* (indicated by the hatched area) that consists of an emulated memory that is established by the trusted VMM. The guest OS has complete control over the emulated memory. Our goal is to execute trusted guest applications within this untrusted zone by leveraging the VMM.

Before the guest OS boots, the VMM establishes a segment of protected physical memory that is beyond the reach of the guest OS. When a guest application runs, the VMM allocates a portion of this memory to store verification and reconstruction data (e.g., hashes, redundant blocks) that is specific to that application. As the system runs, it alternates between *user mode*, in which the application executes, and *kernel mode*, in which the guest OS has control of the system.

Our basic approach is to use the VMM to detect these mode switches and perform some additional work. When the switch is from user mode to kernel mode, we take a snapshot of the application and encode the application to generate some redundant data. Later, when the mode switches back from kernel mode to user mode, we check the integrity of the application's memory image and, if necessary, use the redundant data to repair any corruption that has occurred.

As a part of the encoding process, the VMM generates a small amount of randomized initialization data before the guest OS boots. As this data is inaccessible to the guest OS or applications, and is randomized at runtime, it thwarts static attack strategies. Also, while our proof-of-concept implementation uses a VMM that executes within a host OS, our architecture also applies to cases when there is no host OS, and the VMM executes on bare hardware. This is how Xen, for example, operates. The advantage of eliminating the host OS is that it reduces the trusted code base.

An important aspect of Fig. 2 is the delineation of the trust barrier. As the untrusted guest OS has full access to the emulated memory space, we make the explicit assumption that everything inside the dotted line is untrusted. Furthermore, the guest OS has access to external devices, such as a hard drive. This access has a direct implication on our system design, as the guest OS may swap a page of emulated memory to the hard drive in response to a *page fault*. However, as we will explain in Section 3.2, the VMM can detect the presence of such an interrupt and monitor the corresponding write. In this way, the VMM can continue to monitor the contents of memory, even if they get swapped out to disk.

3.2 Attack Model

The aim of our system is to provide a resilient execution environment for *trusted* guest applications. That is, we assume the protected application has undergone extensive analysis, possibly including formal methods, to verify that the code is immune from common vulnerabilities, such as buffer overflows. As such, we do not consider any attacks exploiting the application itself. However, the rest of the virtual environment, including the guest OS and other guest applications, is generally untrusted. In Section 4, we describe minor exceptions to this assumption. We consider the underlying host OS (if there is one, which is not true for VMMs like Xen) and the VMM to be trusted (assumptions that are common in the literature on virtualization-based security).

As a description of a sample attack, consider a trusted application running on a guest platform that also has a vulnerable network-facing application, such as an email client. Our model is that an attacker exploits a vulnerability in the untrusted application to inject code in the OS. The injected code then corrupts that application's page tables to point to the trusted application, and then modifies the trusted application's memory contents. Thus, the memory corruption originates from a source outside of the trusted application. We also assume that the attacker corrupts *only a small portion of the trusted application*. This is consistent with scenarios where attackers aim to remain stealthy, while causing the application to deviate from the correct execution flow.

4 System Details

Our goal is to recover from memory corruption attacks by applying error-correcting codes to the contents of memory. When an application is running in user mode, the CPU has a current privilege level (CPL) of three; when the OS kernel takes over, the CPL switches to 0. Inside the VMM, we detect whenever the CPL switches from user mode to kernel mode (i.e., the CPL switches from three to zero). When this occurs, we take, in essence, a snapshot of the current application memory image. If the kernel is malicious and then corrupts the contents of memory, we attempt to repair the damage by restoring the snapshot when the application continues processing. Before describing the details of how our solution works, we provide a small amount of relevant background material describing the x86 memory layout.

4.1 x86 Memory Layout

Fig. 3 illustrates the key components of the x86 memory layout and virtual memory addressing techniques for two applications.[2] The virtual memory space of each application is divided into a sequence of pages, typically 4 KB each. When an application references a virtual memory address, the virtual page must be translated into a *frame* in physical memory. This translation process is handled by the hardware *memory mapping unit* (MMU).

The address of the frame is primarily found using two techniques. First, a *translation lookaside buffer* (TLB), which resides in high-speed cache, may contain the frame address

2. Technically, we should make a distinction between the *application*, which is a user-level abstraction, and the *process*, which is a unique OS-level thread of execution. Moreover, an application may consist of multiple processes. However, since our discussion centers on the idea of a *trusted application*, we will primarily use that term.

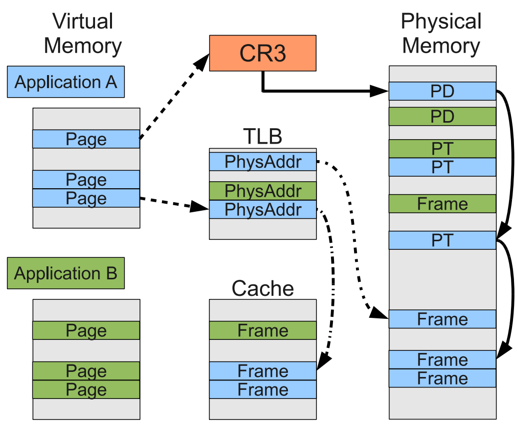

Fig. 3. Overview of the x86 memory layout, including page tables and page directories.

if the page has been recently accessed. If the TLB does not contain the address, then a *page walk* is initiated. In a page walk, the control register CR3 is used to locate the page directory (PD in the figure), which is an array of entries for locating page tables (PT in the figure). Depending on the page size, one or more page tables will be traversed. Finally, the physical address for the frame will be found as an entry in the page table. For increased performance, any frame, including the page directory and page tables, may be stored in a high-speed cache.

Fig. 3 does not show how *demand paging* influences memory, though. In demand paging, if a page has not been previously accessed, then there is no corresponding physical frame. Rather, when the first access occurs, a *page fault* causes the OS kernel to load the data from a *backing store*, typically a hard drive. Once the frame is loaded and the page tables have been updated, the application continues executing as expected.

In our solution, we have integrated LT and Reed-Solomon codes into the physical memory access of the VMM, and we have accomplished this using a technique similar to demand paging. Specifically, we decode a frame corresponding to user-mode applications only when the application actually references that page. When control of the CPU is taken away from the application, we then encode all referenced frames and store checksums to ensure the integrity of each frame. We also store redundant information to help recovery. After the encoding, the OS kernel executes. If the kernel attempts to corrupt the application in any way, our decoding techniques can detect the tampering. Furthermore, we provide probabilistic guarantees that our techniques can *repair* the memory image even if corruption has occurred. Note that our techniques are operating directly on the frames themselves, regardless of whether they are stored in main memory or in a cache.

4.2 Memory Encoding and Decoding

Fig. 4 shows a sampling of some key events during execution of our VMM-based protection mechanism. In a multitasking system, user mode applications execute for small periods of time, called *quantums*. The quantum ends when a hardware interrupt occurs or the application issues a *system call*, which is a request for a service from the OS kernel. When this occurs, the CPU performs a *mode switch* from user mode (CPL 3) to kernel mode (CPL 0). As this

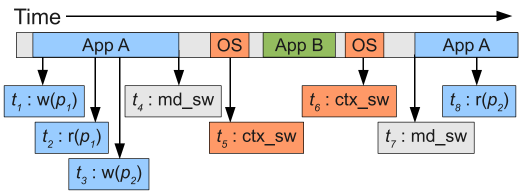

Fig. 4. Key events as execution progresses. Unlabeled time in gray indicates VMM execution.

switch occurs in hardware, the VMM is able to detect the switch and interrupt processing. Another mode switch also occurs when returning to user mode. The events at times t_4 and t_7, denoted *md_sw*, correspond to mode switches. When the OS kernel itself is executing, it may issue a *context switch* (*ctx_sw*), as indicated at times t_5 and t_6. In a context switch, the kernel performs a number of tasks, including updating the CR3 register for a different application. The gray time periods in Fig. 4 correspond to the execution of the VMM.

The influence of demand paging on our design becomes clear when considering the events at times t_1, t_2, t_3, and t_8. When a quantum begins (at a mode switch from CPL 0 to 3), the application's memory image is in an encoded form. However, we only decode a frame *when it is first accessed*. So, at time t_1, application A performs a write to page p_1 (which we assume is the first reference to p_1 for illustration). When the frame is located, the VMM interrupts processing to decode the frame. Then, at time t_2, the VMM does not have to do any work, as p_1 is already decoded. At time t_3, p_2 must be decoded.

When the mode switch occurs at time t_4, the VMM must encode all pages that have been decoded during the previous quantum. In this figure, that means p_1 and p_2 must be encoded. There is one exception to this rule, which we will discuss in Section 4.5. The system will then continue processing as usual until the mode switch at t_7. Just like before, we do nothing immediately. However, at t_8, the frame for page p_2 must again be decoded, because it has not previously been accessed during this quantum.

The advantages of our approach are twofold. First, by performing the decoding on-demand, we significantly reduce the overhead our system imposes, as we will show experimentally in Section 5. Second, by performing the encoding of frames at the mode switches, we are ensuring that *no kernel instruction will execute before encoding occurs*. Thus, if the kernel has become corrupted, the VMM successfully protects the application from attack. We implemented two versions of our design, with LT codes and Reed-Solomon codes, respectively. We provide the details next.

4.2.1 LT Codes-Based Approach

Algorithm 1 describes the encoding process using LT codes. Each frame is partitioned (routine *Partition*) by interpreting it as a sequential array of K blocks of equal size. These blocks are encoded (routine *Encode*) into $N > K$ blocks according to a bipartite graph (see Appendix A in the online supplementary material). The first K encoded blocks are copied back into the frame in memory, whereas the remaining $N - K$ are stored in the VMM protected space. The VMM also stores the hash of each of the first K blocks as well as the hash of the original decoded frame.

Algorithm 1: EncodeMemoryImageWithLT

```
Input: $ReferencedFrames$: an array containing the frame
       numbers referenced during the quantum
$i \leftarrow 0$ ;
while ($i < ReferencedFrames.length$) do
    $FrameNumber \leftarrow ReferencedFrames[i]$;
    $Frame \leftarrow$ GetPhysicalFrame($FrameNumber$);
    $EncodingData.hash \leftarrow$ Hash($Frame$) ;
    $KBlocks \leftarrow$ Partition($Frame$);
    $NBlocks \leftarrow$ Encode($KBlocks$);
    for $j := 0$ to $K - 1$ do
        $EncodingData.blockhash[j] \leftarrow$ Hash($NBlocks[j]$);
        write $NBlocks[j]$ to $Frame$ ;
    for $j := K$ to $N - 1$ do
        $EncodingData.blocks[j - K] \leftarrow NBlocks[j]$;
    $StoredData[FrameNumber] \leftarrow EncodingData$ ;
    $i \leftarrow i + 1$ ;
```

The VMM stores a global array of data structures, denoted as *StoredData*. Each data structure stores the extra $N - K$ blocks plus the hashes of the K blocks stored in OS-accessible memory. The structure *EncodingData* consists of *EncodingData.blocks* (the array of $N - K$ extra blocks), *EncodingData.blockhash* (the array of hashes for the K blocks in memory), and *EncodingData.hash* (the hash of the entire frame). Its counterpart *DecodingData* is used for decoding in Algorithm 2 (the *Decode* routine is presented in Appendix A, which is available in the online supplemental material).

Algorithm 2: DecodeSingleReferencedPageWithLT

```
Input: $FrameNumber$: the index of the frame to be
       decoded
$Frame \leftarrow$ GetPhysicalFrame($FrameNumber$);
$NBlocks \leftarrow$ Partition($Frame$);
$DecodingData \leftarrow StoredData[FrameNumber]$;
$DecodeBlocks \leftarrow \emptyset$;
for $i := 0$ to $K - 1$ do
    if $Hash(NBlocks[i]) = DecodingData.blockhash[i]$
    then
        append $NBlocks[i]$ to $DecodeBlocks$ ;
for $i := 0$ to $N - K - 1$ do
    append $DecodingData.blocks[i]$ to $DecodeBlocks$;
$KBlocks \leftarrow$ Decode($DecodeBlocks$);
write $KBlocks$ to $Frame$;
if $DecodingData.hash \neq Hash(Frame)$ then
    abort with error;
```

4.2.2 Reed-Solomon Codes-Based Approach

The other implementation of our approach was to use Reed-Solomon codes. The disadvantage of Reed-Solomon codes is that the encoding and decoding process takes longer (see Section 5). However, Reed-Solomon allows us to keep the contents of main memory intact. Specifically, a Reed-Solomon code takes an array of k symbols, and appends it with $n - k$ symbols, called the *syndrome*. At decoding time, Reed-Solomon can then correct up to $(n - k)/2$ corrupted symbols to recover the original data. As such, we only need to store the syndrome, and we do not need to write anything back to main memory.

The fact that main memory is not modified during the encoding process allows for an optimization: if the page was only referenced for read operations (i.e., it has not been modified), then we do not have to perform the encoding of that frame. Furthermore, we only need to store a single hash for the entire frame. At decoding time, if the hash of the frame matches the stored value, there is no need to decode. These optimizations make the Reed-Solomon approach very advantageous when most memory accesses are simply read operations.

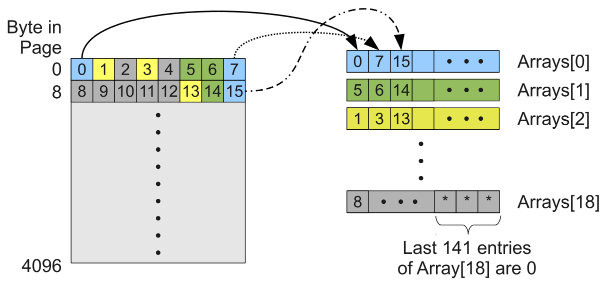

Fig. 5. Randomized mapping of bytes in a page to 19 arrays for Reed-Solomon (255, 223) encoding. Each Arrays[i] has 223 entries. Since Arrays [18] only receives 82 bytes from the page, that array is padded with values of 0 to get a length of 223 bytes.

There is one other important consideration for the Reed-Solomon implementation that did not arise in the LT code version. Specifically, we do not want to partition the page simply into an array of arrays. If we were to do so, the adversary would only need to corrupt $\lceil (n - k)/2 \rceil + 1$ consecutive bytes to successfully corrupt the frame. That is, when the Reed-Solomon parameters are (255, 223), corrupting 17 bytes would be sufficient for a successful attack (see Section 2).

To mitigate this threat, the VMM generates a random mapping for bytes within a frame at runtime. For a 4 KB frame, each byte is assigned a random value from 0 up to $\lceil 4{,}096/k \rceil - 1$, where this value indicates one of the arrays. So, if (255, 223) codes are used, there are $\lceil 4{,}096/223 \rceil = 19$ arrays, and each byte would be assigned to one of the 19 arrays. However, the randomization ensures that the first 18 arrays will each have 223 bytes assigned to them. The last array will get the remaining 82 bytes, and will be padded with 0 to also have length 223. Hence, when encoding or decoding, the *Partition* and *Departition* functions loop through the 4,096 bytes of the frame, copying each byte to the corresponding array. Fig. 5 illustrates the randomized mapping for the (255, 223) encoding. As we will show later in Section 5, this randomization reduces the attack success rate. The attacker must now corrupt 305 bytes under the (255, 223) parameters to guarantee success. Specifically, there are 19 arrays, and each can tolerate 16 corrupt symbols; 305 corrupt bytes guarantees that at least one array has 17 corrupt symbols.

Algorithm 3 shows the algorithm for encoding the referenced frames under the Reed-Solomon approach. As before, *StoredData* is an array of data structures stored in the protected VMM space. In this case, *StoredData* consists of a single SHA-1 value *StoredData.hash*, an array of syndromes *StoredData.syndromes*, and a flag to indicate the page has been modified.[3] When the mode switch occurs, the VMM checks to see if the page has been modified by checking the flag. If the page has not been modified, there is no encoding

3. We could actually use the dirty bit in the corresponding page table entry for this purpose, but that would require doing a page walk. Hence, we chose to sacrifice a small amount of space to optimize for speed.

that is needed. If the flag indicates a write has occurred, we hash the page to see if the write actually changed the contents (i.e., it did not write a value that matched what was already stored). Only if the page has actually modified do we partition the frame and create the syndromes.

Algorithm 3: EncodeMemoryImageWithRS

```
Input: ReferencedFrames: an array containing the frame
       numbers referenced during the quantum
$i \leftarrow 0$ ;
while ($i < ReferencedFrames.length$) do
    $FrameNumber \leftarrow ReferencedFrames[i]$
    $Frame \leftarrow$ GetPhysicalFrame($FrameNumber$)
    $EncodingData \leftarrow StoredData[FrameNumber]$
    if $EncodingData.flag = 0$ then
        return ;
    $h \leftarrow$ Hash($Frame$);
    if $EncodingData.hash = h$ then
        return ;
    $EncodingData.hash \leftarrow h$;
    $Arrays \leftarrow$ Partition($Frame$);
    pad $Arrays[Arrays.length - 1]$ with 0 for correct
    length ;
    for $j := 0$ to $Arrays.length$ do
        $EncodingData.syndrome \leftarrow$ Encode($Arrays[j]$);
    $i \leftarrow i + 1$ ;
```

Algorithm 4 describes the algorithm for decoding the frame under the Reed-Solomon approach. Note the optimization in lines 3-5: if the page's hash matches the stored hash value, then there is no need to do any decoding.

Algorithm 4: DecodeSingleReferencedPageWithRS

```
Input: FrameNumber: the index of the frame to be
       decoded
$Frame \leftarrow$ GetPhysicalFrame($FrameNumber$);
$DecodingData \leftarrow StoredData[FrameNumber]$;
if $DecodingData.hash = Hash(Frame)$ then
    return ;
$Arrays \leftarrow$ Partition($Frame$);
pad $Arrays[Arrays.length - 1]$ with 0 for correct length ;
for $i := 0$ to $Arrays.length$ do
    append $DecodingData.syndrome[i]$ to $Arrays[i]$;
    $Decoded \leftarrow$ Decode($Arrays[i]$);
    Departition($Decoded$);
if $DecodingData.hash \neq Hash(Frame)$ then
    abort with error ;
```

4.3 Storage of Recovery Information

Fig. 6 shows the structure of data storage for a single page under both schemes. In the LT approach, we assume $K = 16$ and $N = 18$. The column on the left shows the original contents of memory. When the data is encoded, the contents of physical memory are then overwritten with the encoded blocks. The additional two blocks are also stored in the VMM, along with the hashes of the blocks stored in physical memory. A hash of the original (unencoded) data is stored for an integrity check. In this figure, we do not show the encoding graph[4] (120 bytes), as all pages share the same graph.

4. For details about the encoding graph concept, please see Appendix A, which is available in the online supplemental material.

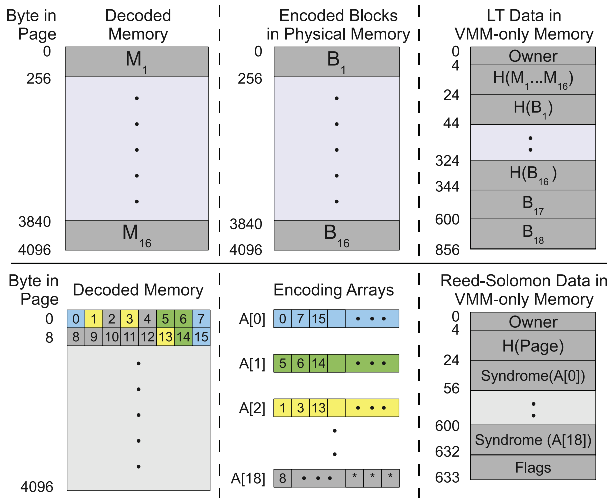

Fig. 6. Decoded memory, intermediate and stored data for LT (top) and Reed-Solomon (bottom).

In the Reed-Solomon scheme, we assume the parameters are (255, 223). To generate the encoding data, the original frame (on the left) is partitioned into 19 arrays of 223 bytes each. The last array consists of 82 bytes from the frame, followed by a padding of 141 bytes with the value 0. For each array, we generate and store a 32-byte syndrome. The one-byte flag is used to indicate that the page has been modified, and the hash of the original (unencoded) data is stored for an integrity check. In this figure, we do not show the page-to-array mapping, which is simply an array of 4,096 bytes.

4.4 Implementation Challenges

Integrating memory encoding into a real system requires addressing a number of issues. First, the VMM must become aware of the OS paging procedure. That is, when the guest OS kernel responds to a page fault, it may choose to swap an application page to disk. Once the data is on the disk, it is beyond the protection of our memory encoding scheme and subject to attack. However, as page faults trigger a hardware interrupt, the VMM can detect this occurrence and monitor for the OS initiating the transfer of a page to the backing store. Later, when that same data is swapped back into memory (as the result of another page fault), the VMM can update its data structures to map the encoding data to the new frame. If the data was corrupted while on disk, the hash will not match when the frame is later decoded.

The next concern is the preservation of state values contained in CPU registers. Specifically, when a mode switch occurs, the registers are storing the current values of some application variables. If the kernel initiates a context switch, it pushes those values onto the application's stack for preservation. When another context switch occurs to return to the original application, these values are popped off the stack and restored to the registers. If the OS is corrupt, it could modify the values stored in the registers before the mode switch returns control to the application. To prevent this, the VMM, in essence, duplicates the context switch work. We use the value stored in the CR3, which is unique to each application,[5] to locate a storage place for the state data.

The last, and most critical challenge, is the distinction between legitimate page modifications and possible attacks from the kernel. Specifically, the main purpose of the OS kernel is to provide user-mode applications with access to hardware resources, such as the network card or hard drive. This access is granted through the *system call* interface. In order for the application to receive the requested data, the kernel must be able to write to a memory location in the application's memory space, including the stack. That means that, when the mode switch returns control to the application, the hash of those frames will not be correct.

To accommodate this, two techniques are required. First, the frame storing the top of the stack will not be encoded. Rather, the VMM will monitor that frame to ensure that the only changes made are appropriate for responding to a system call (e.g., the memory storing the index of the desired system call will be popped by the kernel). Second, we must reserve a sequence of pages in the virtual address space that will not be encoded. The kernel can write data to these pages, then the application can copy the data to an appropriate location, such as its heap. These two techniques allow the kernel to perform legitimate memory writes, while preventing full access to the memory space.

4.5 On Shared Memory

While the preceding section described techniques for preventing the kernel from writing to the protected application, indirect corruption is still possible. That is, the kernel could collude with a malicious user-mode application $\mathcal{A}$ and set up $\mathcal{A}$'s page tables to write to a protected application. We combat this threat by storing the *owner* of the page, and enforcing a rule of *no bidirectional shared memory writes*. Consider two applications A and B that need to exchange data. If A wishes to send data to B, it would store the data in a page owned by A. B could then read that data, but could not modify it. If B needed to respond in some way, it would write that data to a page that B owns and A could later read. Thus, legitimate user-mode applications that need to share data would have to cooperate accordingly.

As for enforcement, the VMM stores information to identify the owner of a page. Specifically, the owner is identified by the CR3 value for the process. When the process first references a page within a quantum, the VMM makes a copy of the original encoded data. Later, when the time quantum expires and the VMM encodes the process's modified pages, it first checks to see if that process is the owner of that particular frame by comparing the CR3 value with the stored owner data. If there is a mismatch, the process is trying to write to a page it does not own, which is forbidden. Consequently, the VMM restores the original encoded data from its copy. While this technique places a burden on developers to avoid bidirectional shared memory, it prevents an indirect corruption by the kernel. Note, however, that this technique does not prevent copy-on-write shared memory, as the kernel must allocate a new page when a write is attempted.

5. In theory, the CR3 could change if the application's page directory is swapped out. This happens rarely in practice, though. To be thorough, the VMM, which is aware of the page directories, can track this and react accordingly.

4.6 Threat Analysis

Our stated goal is to protect trusted guest applications from memory corruption attacks launched by the guest OS or other guest applications. We provide this protection through the combination of the encoding scheme and verifying the integrity of encoded blocks with a cryptographic hash function. While our design offers a robust layer of protection, we have identified a number of remaining threats. First, our system must trust the software that exists outside the trust boundary (Fig. 2). Specifically, the VMM must be considered trusted. Also, if a host OS exists below the VMM, it must also be trusted. This also means that if the host OS swaps a frame out to disk, an attacker with full access to the disk could potentially attack the system. Thus, eliminating the host OS (i.e., using a VMM that runs on bare hardware) helps to reduce the assumption of trust.

Corruption attacks. In order for an attacker to corrupt a guest application successfully, he would first have to construct a collision under the hash algorithm. To complicate that search, the collision must be the same size as the encoded block (in the LT approach) or the full page (in the Reed-Solomon approach). If, at any point, the required hash does not match, that portion of memory may be discarded. Furthermore, if the final page does not hash correctly (i.e., our decoding failed), the system will simply abort. While the use of SHA-1 makes finding a collision difficult (that is, nearly impossible in practice), the runtime randomization of the LT encoding graph and the Reed-Solomon partition map make the attack even more difficult. As these data structures are inaccessible to the guest OS and applications, the attacker would have to resort to guessing. Thus, we find successful corruption attacks from other guest applications to be extremely implausible, though possible in theory.

Denial of service. While our system assumes that the guest OS may be compromised, we are assuming that it is still somewhat functional. That is, if the attackers goal is merely to shut down the system, she could corrupt the OS beyond repair. This attack is outside our scope, as we focus on scenarios where the attacker's goal is to keep the system working in a compromised fashion. A more subtle attack could aim to prevent the OS from scheduling the trusted application. To defend against this attack, we suggest deploying a technique similar to a watchdog timer. That is, the application may send a report to an external monitor at least once within a certain time period. If the application has not been running, the fact that the monitor does not receive the report will indicate that this attack has occurred. At that point, external administration would be required to restart or repair the system. To prevent the guest OS from forging such reports, cryptographic techniques can be applied to network data, as described below.

VMM exploits. A software VMM is subject to vulnerabilities just like any other piece of code. Obviously, if the adversary can corrupt the VMM, then our system offers no guarantees of security. However, a number of projects have made great strides toward protecting the VMM from corruption [11], [12], [13]. As such, we simply rely on an assumption of trust in the VMM.

Application vulnerabilities. Our design assumes that the protected application is vulnerability-free. Clearly, such an assumption cannot be made in general, as buffer overflows and other vulnerabilities remain a common problem. However, many environments require trusted applications that have undergone formal analysis and verification. Our design offers these applications a resilient execution environment that protects the application from *external* threats.

Corrupted network data. If the critical application sends or receives data across a network interface, the guest OS kernel has an opportunity to modify this data in transit. Specifically, access to the network card is set up by the OS via a system call. In the case of verifying received data, a public key could be hard-coded into the application without incurring a significant threat. Securing private or symmetric keys, though, is not possible in our current approach, as the guest OS has full access to read that application's memory space and would be able to forge messages easily. Instead, for applications that require secret keys, we propose combining our approach on top of existing solutions, such as Overshadow [5][6] or a virtual trusted platform module (TPM) [14]. By utilizing other protection mechanisms, the application could have secured access to a secret key as needed.

4.7 Fail-Safe Recovery

Recall that our goal was to detect corruption in all cases and to provide a mechanism to repair the application dynamically without interrupting service. Clearly, our use of cryptographic hash functions to authenticate the integrity of each memory page before it is accessed accomplishes the former.[7] We have also proposed a scheme that provides a *probabilistic guarantee* of the latter. That is, our system, as described above, may not be able to recover transparently, especially if the amount of corruption within a single page is large.

Observe that, if the recovery fails, the VMM is immediately aware of the failure, and knows exactly which page in the application's linear address space is corrupted. As the trusted component, the VMM would have full access to replace the memory image as necessary. If the corrupted page consists of code or read-only data, the VMM can extract the relevant portion of the application from a protected local storage that is inaccessible to the guest OS. This procedure would allow the application to continue processing as before.

The more problematic failure is if the corrupted page contains data that has been modified at runtime. Clearly, the VMM has insufficient information to initiate a transparent local recovery. Instead, the VMM would initiate a remote recovery protocol with a trusted server. Note that this communication channel would not be threatened by the guest OS, as the VMM would have direct control of the network interface. The VMM would then reinstate the memory image saved at the last checkpoint, as shown in Fig. 1a. Note that, in that case part of the execution must be rolled back.

6. Recall that Overshadow ensures confidentiality and integrity by encrypting the memory image. However, our goals are integrity and *availability*, the latter of which is not addressed by Overshadow. As such, we see our schemes as complementary, and believe they could be combined for even greater protection.

7. As of this writing, collision attacks in SHA-1 are not practical. If this threat is a concern, one could upgrade the hash function to SHA-256 or other stronger functions.

5 Implementation and Evaluation

To provide a proof-of-concept, we have implemented a preliminary prototype to measure the performance impact of our system design. We have modified the source code for version 0.9.0 of the QEMU hardware emulator [15]. We adapted the LT code library developed by Uyeda et al. [16] as well as the Reed-Solomon implementation by Rockliff.[8] Whenever an operation causes a switch to kernel mode (i.e., the CPL switches from 3 to 0), we encode the most recently used pages as described in Section 4. Decoding is done in an on-demand manner, whenever a page is first referenced during a quantum.

5.1 Performance Overhead

Our test platform consisted of a 2.26 GHz Intel Core2 Duo CPU with 3 GB of 667 MHz memory, running Ubuntu 10.04 (Lucid Lynx), with version 2.6.32-29 of the Linux kernel as the host OS. Our VMM was an adaptation of QEMU version 0.9.0, and the guest virtual machine was running Redhat 9 with version 2.6.20 of the Linux kernel on 128 MB of emulated physical memory.

Our prototype consisted of three implementations. The first two integrated LT codes and Reed-Solomon codes as described previously. The third consisted of the naïve implementation, in which each page of the user-mode application is simply cloned, rather than encoded. While simply cloning the page adds almost no performance overhead, this third implementation also included the SHA-1 calculation to check the integrity of the page. Incorporating this calculation allowed us to determine how much overhead resulted from the encoding scheme itself as compared to the integrity check.

For both the clone and the Reed-Solomon cases, we strove to ensure that we measured the difference that would result between doing a full frame encoding/decoding as compared to one that did not require encoding/decoding. Recall that if the page has not been modified (i.e., the current hash matches the previous), then no additional work is required. To measure the difference, we used a ratio that ensured approximately one percent of all pages referenced required the full processing. The remaining 99 percent required only the initial SHA-1 hash calculation.

Fig. 7 summarizes the overall performance impact for the *nbench* benchmark suite [17]. As a baseline figure, we measured an average time for a normal mode switch to be about $3.2\ \mu s$. Clearly, the average total time per mode switch (T/MS) for all three prototype implementations is much larger than this baseline. However, comparing the clone and Reed-Solomon implementations shows that the SHA-1 calculations account for the majority of the overhead. This impact is exacerbated in the LT case, as 16 hashes must be computed for each page (one for each LT block in unprotected memory). The remaining two metrics are the average time spent encoding/cloning (T/EF) or decoding (T/DF) a single frame. The LT scheme has the highest average time to process a frame, because every frame must be encoded and decoded; in contrast, only one percent of the Reed-Solomon and cloning frames must be fully processed.

8. Rockliff's Reed-Solomon code software can be downloaded at http://www.eccpage.com/.

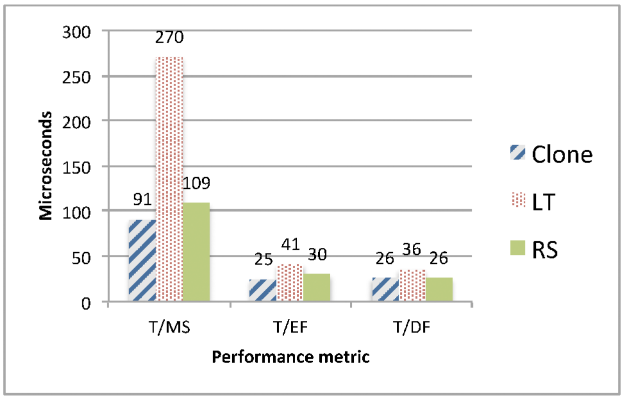

Fig. 7. Summary of encoding performance impact.

Fig. 8 provides more insight into the Reed-Solomon and clone implementations. The encoding measurements are broken down into those that require full processing (T/EF/F for encoding, T/DF/F for decoding), and those that can have the encoding/decoding skipped (T/EF/S, T/DF/S) because the hashes matched. Recall that our experiments simulated a rate that one percent of pages are corrupted; consequently, while the Reed-Solomon encoding and decoding require significant time, this rarely occurs, yielding the low average time per frame shown in Fig. 7.

5.2 Recovery Success and Memory Overhead

Recall that both the LT and Reed-Solomon schemes provide only a probabilistic guarantee of data recovery. Fig. 9 summarizes the success rates for both schemes, using $K = 16$ with various sizes of N for the LT codes, and Reed-Solomon codes with $n = 255$ and various sizes of k.

As our LT prototype was designed so that only 16 of the encoded blocks are to be stored in physical memory, we limited our evaluation to cases where some of these 16 blocks are corrupted. The results shown in Fig. 9 are the result of executing the LT encoding on a random block of 4,096 bytes and randomly selecting blocks to discard. For each number of discarded blocks, we executed the encoding and decoding 10,000 times. For $N = 18$, it is encouraging that we have more than a 50 percent chance of successful recovery for up to two corrupted blocks. As N increases, so do the chances of recovery. For $N = 20$, three blocks can be corrupted while still providing more than a 30 percent chance of recovery. The tradeoff, though, is that $N = 18$ requires less than 20 percent memory overhead (as shown in Fig. 10), while $N = 20$ would require approximately 32 percent memory overhead (to accommodate for the two additional blocks).

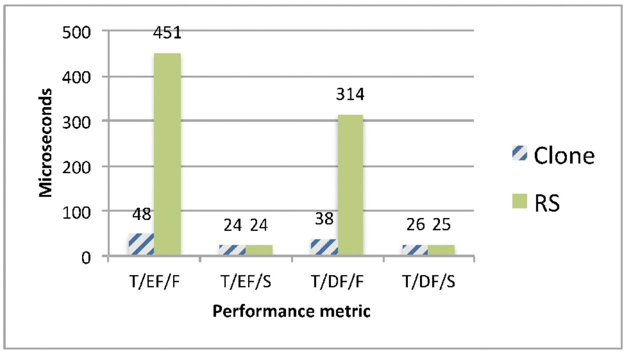

Fig. 8. Detailed performance impact of the Reed-Solomon and clone implementations.

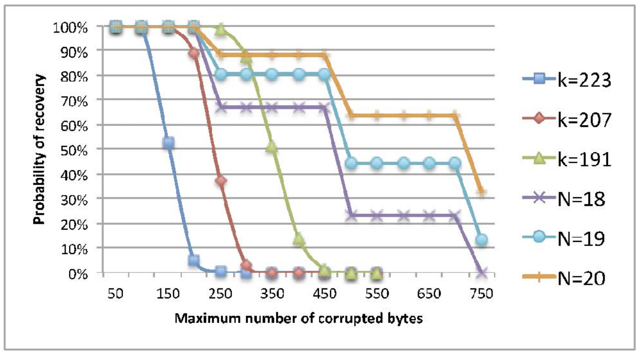

Fig. 9. Rate of successful data recovery for LT codes for $K = 16$ and various sizes of N.

To evaluate the probability of successful recovery with Reed-Solomon codes, we performed a probabilistic analysis, where the attacker randomly selects and corrupts a number of bytes. To simplify the analysis, we used a generating function based on the following polynomial:

$$P(z) = \left(\sum_{j=0}^{(n-k)/2} z^j \right)^{\left\lceil \frac{4096}{k} \right\rceil}.$$

When this polynomial is expanded, the ratio of the coefficient of z^i to the sum of all coefficients provides an approximation of the probability that, given a random choice of i bytes, the attacker has successfully picked more than $(n-k)/2$ from one of the arrays. That is, this ratio is the probability that the attacker has successfully corrupted the page beyond our recovery technique. Using this technique, Fig. 9 shows the cumulative probability that our approach can successfully recover given various parameters for k and the number of bytes corrupted. For instance, using Reed-Solomon (255, 223), if the attacker corrupts less than 150 bytes in a single frame, our system can repair the damage with a probability of 52.78 percent.

We selected the parameters for LT and Reed-Solomon codes based on the amount of extra memory required to store the extra data. Figs. 10 and 11 summarize the extra memory required for each page of the user-mode application. Note that, in the memory cloning version of our prototype, the overhead would be more than 100 percent, as the full page, its hash, and the owner field would all be stored in the VMM-protected memory.

Value of K	Hash overhead	Owner, extra blocks	Total overhead	Percent of memory
8	180	1028	1208	29.49 %
16	340	516	856	20.90 %
32	660	260	920	22.46 %

Fig. 10. LT storage overhead per page (in bytes) for various sizes of K, assuming $N = K + 2$.

(n, k)	Hash, flag, owner	Syn-drome	Total overhead	Percent of memory
(255,223)	25	608	633	15.45 %
(255,207)	25	960	985	24.05 %
(255,191)	25	1408	1433	34.99 %

Fig. 11. Reed-Solomon overhead (in bytes) per page.

In the case of LT codes, Fig. 10 summarizes the extra memory required to store the hashes and extra encoded blocks for various sizes of K. (For completeness, we note that the storage of the encoding graph (120 bytes) is trivial, since it is shared for all frames.) As shown in the figure, using fewer than $K = 16$ blocks would increase the memory overhead to more than 20 percent. On the other hand, using more than 16 blocks increases both the storage and the performance overhead, as the VMM would be required to devote more resources to computing and storing the hashes of each block. Thus, $K = 16$ offers a balance that minimizes both the memory and performance overhead.

Fig. 11 summarizes the storage overhead per page for the Reed-Solomon version. Note that $n = 255$ is the only feasible first parameter, as $n = 2^m - 1$, where m is the number of bits in each symbol. Accommodating $m \neq 8$ would require too many bit manipulations to provide a feasible approach. For the second parameter, we consider k values of 223, 207, and 191, which allow for the correction of 16, 24, and 32 byte corruptions (per array), respectively. Obviously, as k decreases, we can provide better probability for successful recovery. However, only $k = 223$ allows us to keep the memory overhead below 20 percent.

6 Related Work

Authentication of executed code has received a lot of attention in recent years, due to novel applications such as outsourcing of services. Flicker [11] uses a trusted platform module to prove to a remote party that a certain sequence of instructions is properly executed. An accumulated hash of a block of instructions is computed by a trusted hardware module, and the hash is signed with a key that is kept in trusted storage. In the Patagonix [18] system, the objective is to prevent execution of unauthorized applications. To that extent, the (trusted) OS maintains a database of hashes for the code and static data segments of known applications. Upon starting a new process, the OS checks if the executable's hash matches one of the values stored in the database, otherwise execution is prohibited. Note that, the setting in Patagonix is quite different from ours, since the OS is trusted. Furthermore, Patagonix does not address memory corruption that occurs at runtime.

Several solutions for secure code execution based on VMMs have been proposed. The Terra system [4] provides critical applications with their own virtual machine, including an application-specific software stack. However, such a solution may not be scalable, and may limit interoperability. HookSafe [19] relies on a trusted VMM to guarantee the integrity of system call hooks that are used by applications. SecVisor [20] virtualizes hardware page protections to ensure the integrity of a commodity OS. XOMOS [21] proposes building an OS to rely on a trusted processor architecture to provide kernel integrity; the authors emulated the behavior in a trusted VMM, which could be used in place of the hardware. Similarly, SecureBit [22] proposes the use of external protected memory to defend against buffer overflows; like XOMOS, the authors emulated their design in a trusted VMM. These approaches either focus on protecting the integrity of the OS or make no attempt to recover after a memory corruption is detected. In contrast, our work focuses on creating an environment in which the application can be repaired and allowed to continue, even if the underlying OS is corrupt.

A number of other techniques have been proposed for resilient processing. For instance, checkpointing [23], [24], [25], returns the execution to a known, good state if a corruption occurs. Similarly, speculative execution [26], [27] performs a number of processing steps, and only commits to the results if they are correct. Some systems leverage multicore environments [28], [29], [30] and parallel execution [31] in an attempt to detect when an application has been corrupted. One such approach [32] will also resurrect a corrupted process automatically. CuPIDS [33] uses a physically separate coprocessor to perform monitoring, similar to VMM-based approaches. Another interesting approach is to simply ignore faults that would lead to a crash and continue processing [34]. While all of these approaches have their merits, they do not consider the case where the OS itself is malicious.

Closest to our work is the Overshadow [5] system, where a VMM prevents a malicious or compromised OS from accessing a protected application's memory space. The VMM encrypts the memory image of the application right after the corresponding process relinquishes the CPU, and before any other (untrusted) code gets the chance to execute. A hash with the digest of the legitimate memory contents is stored in trusted storage. However, Overshadow terminates the critical application as soon as corruption is detected. In contrast, we do attempt to recover the correct memory image and resume execution. Furthermore, the use of encryption in Overshadow leads to a more significant performance penalty than our approach. However, if encryption is vital to the protection of the application, our solution can be immediately extended to encrypt memory contents.

The problem of protecting secrecy of data in outsourced applications is addressed in [35] by creating a *Software Privacy-Preserving Platform (SP^3)*. Similar to Overshadow, SP^3 also encrypts sensitive data, but it takes a more flexible approach, where only part of the application memory is encrypted. Furthermore, SP^3 allows policy-based specification of protection domains, and only programs that run within the domain are allowed access to decrypted contents.

In another similar work, NICKLE [6], [36] protects against kernel rootkits by keeping a duplicate image of kernel memory in a protected partition of the physical memory. Whenever an access to the kernel memory is performed, the running copy is compared to the saved copy, and an alarm flag is raised if a mismatch is detected. Similar to Overshadow and to our approach, NICKLE also relies on a trusted VMM. NICKLE only protects the kernel, but not applications running in user space, and the duplicate image must be created at bootstrap time. Furthermore, duplication of memory is not a scalable solution to protect multiple applications. To complicate things further, the memory image of applications changes frequently, whereas NICKLE only protects the kernel code segment which does not change.

Finally, ClearView [37] incorporates machine learning techniques into a VMM to patch running applications for common vulnerabilities without reboot. ClearView requires a learning phase, in which the VMM identifies correlated invariants, i.e., actions that are associated with failures. When an invariant is violated, ClearView automatically generates a patch for the executing process. ClearView is aimed at untrusted applications that may be vulnerable to traditional exploits, such as buffer overflows; ClearView does not protect against corruptions that do not lead to application failures. In contrast, we offer a defensive mechanism to prevent external applications and an untrusted OS from corrupting any portion of a trusted application.

Error-correcting codes (or erasure codes) have been extensively used in communication protocols for lossy channels. Maximum distance separable codes (e.g., Hamming codes) are optimal in terms of reception efficiency, but incur high encoding and decoding complexity (often quadratic to the size of the message). Such codes rely on parity checks, or polynomial oversampling. *Near-optimal* erasure codes tradeoff some additional redundant storage requirements, but achieve fast computation time for encoding and decoding. Luby transform (LT) [10] and Raptor [38] codes fall in this category. We use LT codes in our implementation of the resilient execution environment for critical applications. Although Raptor codes are superior in principle to LT codes, they incur additional implementation complexity that may not be suitable for low-level deployment, such as within a VMM.

7 Conclusion

The current state of modern computer systems requires the execution of trusted applications in untrusted environments. In this work, we have proposed a novel approach for protecting applications against memory corruption attacks by incorporating efficient encoding of memory contents during the context switch procedure. In our approach, when the context switches from guest application A to B, the VMM encodes A's memory image, then decodes B's image after performing an integrity check. Unlike previous schemes, ours is unique in the sense that only ours offers the possibility that *the corrupted application can be repaired and allowed to continue execution.*

We presented both the design and a prototype implementation. Our empirical results show that both the memory and performance overhead imposed by our design is reasonable. One possible direction for future work would be to offer a more fine-grained approach to memory protection. For instance, it may be desirable to protect only critical portions, therefore reducing execution and protected storage overhead. Another direction would be to define the protocols and program analysis techniques for remote application recovery, as discussed in Section 4.7. Next, other types of encoding schemes could be explored. For instance, Raptor codes offer faster performance, but greater implementation complexity, and would be a good choice for additional evaluation. Finally, other VMM-based protection mechanisms offer additional protections that our scheme does not address. For example, Overshadow also provides confidentiality for the application. We view our scheme as complementary to these other solutions, and believe that combining them would create a very strong protection mechanism. Quantifying the costs involved in such a scheme would be an important step in evaluating such a system.

Acknowledgments

The work reported in this paper has been partially supported by Sypris Electronics. Half of the work by M. Kirkpatrick and G. Ghinita was done while at Purdue University.

References

[1] J. Xu and N. Nakka, "Defeating Memory Corruption Attacks via Pointer Taintedness Detection," *Proc. Int'l Conf. Dependable Systems and Networks,* pp. 378-387, 2005.

[2] F-Secure, "Stuxnet Questions and Answers," http://www.f-secure.com/weblog/archives/00002040.html, 2010.

[3] A. Fox and D. Patterson, "Self-Repairing Computers," *Scientific Am.,* vol. 288, pp. 54-61, June 2003.

[4] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A Virtual Machine-Based Platform for Trusted Computing," *Proc. 19th ACM Symp. Operating System Principles (SOSP),* pp. 193-206, 2003.

[5] X. Chen, T. Garfinkel, E.C. Lewis, P. Subrahmanyam, C.A. Waldspurger, D. Boneh, J. Dwoskin, and D.R.K. Ports, "Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems," *Proc. 13th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* 2008.

[6] R. Riley, X. Jiang, and D. Xu, "Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing," *Proc. 11th Int'l Symp. Recent Advances in Intrusion Detection (RAID),* pp. 1-20, 2008.

[7] E.N.M. Elnozahy, L. Alvisi, Y.-M. Wang, and D.B. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," *ACM Computing Surveys,* vol. 34, pp. 375-408, Sept. 2002.

[8] D.J.C. Mackay, "Fountain Codes," *Proc. IEE Comm.,* vol. 152, no. 6, pp. 1062-1068, 2005.

[9] I.S. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields," *J. Soc. for Industrial and Applied Math. (SIAM),* vol. 8, no. 2, pp. 300-304, 1960.

[10] M. Luby, "LT Codes," *Proc. IEEE 43rd Ann. Symp. Foundations of Computer Science (FOCS),* pp. 271-280, 2002.

[11] J.M. McCune, B.J. Parno, A. Perrig, M.K. Reiter, and H. Isozaki, "Flicker: An Execution Infrastructure for TCB Minimization," *ACM SIGOPS Operating System Rev.,* vol. 42, no. 4, pp. 315-328, 2008.

[12] A.M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N.C. Skalsky, "Hypersentry: Enabling Stealthy In-Context Measurement of Hypervisor Integrity," *Proc. 17th ACM Conf. Computer and Comm. Security (CCS),* pp. 38-49, 2010.

[13] Z. Wang and X. Jiang, "HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity," *Proc. IEEE Symp. Security and Privacy,* 2010.

[14] S. Berger, R. Cceres, K.A. Goldman, R. Perez, R. Sailer, and L. Doorn, "vTPM: Virtualizing the Trusted Platform Module," *Proc. 15th Conf. USENIX Security Symp.,* pp. 305-320, 2006.

[15] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," *Proc. USENIX Ann. Technical Conf.,* pp. 41-46, 2005.

[16] F. Uyeda, H. Xia, and A.A. Chien, "Evaluation of a High Performance Erasure Code Implementation," Technical Report CS2004-0798, UCSD Computer Science and Eng., Sept. 2004.

[17] U.F. Mayer, "Linux/unix Nbench," http://www.tux.org/mayer/linux/bmark.html, 2003.

[18] L. Litty, H.A. Lagar-Cavilla, and D. Lie, "Hypervisor Support for Identifying Covertly Executing Binaries," *Proc. 17th USENIX Conf. Security Symp.,* pp. 243-258, 2008.

[19] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering Kernel Rootkits with Lightweight Hook Protection," *Proc. 16th ACM Conf. Computer and Comm. Security (CCS),* pp. 545-554, 2009.
[20] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity Oses," *Proc. 21st ACM SIGOPS Symp. Operating System Principles (SOSP),* pp. 335-350, 2007.
[21] D. Lie, C.A. Thekkath, and M. Horowitz, "Implementing an Untrusted Operating System on Trusted Hardware," *Proc. 19th ACM Symp. Operating System Principles (SOSP),* pp. 178-192, 2003.
[22] K. Piromsopa and R.J. Enbody, "Secure Bit: Transparent, Hardware Buffer-Overflow Protection," *IEEE Trans. Dependable and Secure Computing,* vol. 3, no. 4, pp. 365-376, Oct.-Dec. 2006.
[23] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A.D. Keromytis, "ASSURE: Automatic Software Self-Healing Using Rescue Points," *Proc. 14th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pp. 37-48, 2009.
[24] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: Treating Bugs as Allergies—A Safe Method to Survive Software Failures," *Proc. 20th ACM Symp. Operating System Principles (SOSP),* pp. 235-248, 2005.
[25] S. Sidiroglou, O. Laadan, A.D. Keromytis, and J. Nieh, "Using Rescue Points to Navigate Software Recovery," *Proc. IEEE Symp. Security and Privacy,* pp. 273-280, 2007.
[26] M.E. Locasto, A. Stavrou, G.F. Cretu, and A.D. Keromytis, "From STEM to SEAD: Speculative Execution for Automated Defense," *Proc. USENIX Ann. Technical Conf.,* pp. 1-14, 2007.
[27] S. Sidiroglou, M.E. Locasto, S.W. Boyd, and A.D. Keromytis, "Building A Reactive Immune System for Software Services," *Proc. USENIX Ann. Technical Conf.,* pp. 149-161, 2005.
[28] R. Huang, D.Y. Deng, and G.E. Suh, "Orthrus: Efficient Software Integrity Protection on Multi-Cores," *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pp. 371-384, 2010.
[29] O. Trachsel and T.R. Gross, "Variant-Based Competitive Parallel Execution of Sequential Programs," *Proc. Seventh ACM Int'l Conf. Computing Frontiers,* pp. 197-206, 2010.
[30] B. Salamat, A. Gal, T. Jackson, K. Manivannan, G. Wagner, and M. Franz, "Multi-Variant Program Execution: Using Multi-Core Systems to Defuse Buffer-Overflow Vulnerabilities," *Proc. Int'l Conf. Complex, Intelligent and Software Intensive Systems,* pp. 843-848, 2008.
[31] B. Salamat, T. Jackson, A. Gal, and M. Franz, "Orchestra: Intrusion Detection Using Parallel Execution and Monitoring of Program Variants in User-Space," *Proc. Fourth ACM European Conf. Computer Systems (Eurosys),* pp. 33-46, 2009.
[32] W. Shi, H.-H.S. Lee, L. Falk, and M. Ghosh, "An Integrated Framework for Dependable and Revivable Architectures Using Multicore Processors," *Proc. 33rd Ann. Int'l Symp. Computer Architecture,* pp. 102-113, 2006.
[33] P.D. Williams and E.H. Spafford, "CuPIDS: An Exploration of Highly Focused, Co-Processor-Based Information System Protection," *Computer Networks,* vol. 51, no. 5, pp. 1284-1298, 2007.
[34] M. Rinard, C. Cadar, D. Dumitran, D.M. Roy, T. Leu, and W.S. Beebee Jr., "Enhancing Server Availability and Security Through Failure-Oblivious Computing," *Proc. Sixth Conf. Symp. Operating Systems Design and Implementation (OSDI),* pp. 21-21, 2004.
[35] J. Yang and K.G. Shin, "Using Hypervisor to Provide Data Secrecy for User Applications on a Per-Page Basis," *Proc. Fourth ACM SIGPLAN/SIGOPS Int'l Conf. Virtual Execution Environments (VEE),* pp. 71-80, 2008.
[36] J. Rhee, R. Riley, D. Xu, and X. Jiang, "Defeating Dynamic Data Kernel Rootkit Attacks via VMM-Based Guest-Transparent Monitoring," *Proc. Fifth Int'l Conf. Availability, Reliability and Security (ARES),* 2009.
[37] J.H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M.D. Ernst, and M. Rinard, "Automatically Patching Errors in Deployed Software," *Proc. 22nd ACM SIGOPS Symp. Operating System Principles (SOSP),* pp. 87-102, 2009.
[38] A. Shokrollahi, "Raptor Codes," *IEEE/ACM Trans. Networking,* vol. 14, no. SI, pp. 2551-2567, June 2006.

Michael S. Kirkpatrick received the graduate degrees from Purdue University and Michigen State University. Prior to earning graduate degrees, he spent several years working in the field of semiconductor optical proximity correction for IBM Microelectronics (later Server and Technology Group) in Essex Junction, Vermont. Currently, he is working as an assistant professor of computer science at James Madison University. He has served as a reviewer for a number of top journals and conferences, including *IEEE TDSC*, *IEEE S&P*, *IEEE ICDM*, *ACM SACMAT*, *ACSAC*, *IEEE TVLSI*, *ACM TISSEC*, and *IFCA Financial Cryptography and Data Security*. His main research interests include security engineering, access control, applied cryptography, privacy, and secure operating systems. He is a member of the IEEE Computer Society.

Gabriel Ghinita is an assistant professor with the Department of computer science, University of Massachusetts, Boston. Prior to joining the University of Massachusetts, he was a research associate with the Cyber Center at Purdue University, and a member of the Center for Education and Research in Information Assurance and Security (CERIAS). He served as reviewer for top journals and conferences such as *IEEE TPDS*, *IEEE TKDE*, *IEEE TMC*, *VLDBJ*, *VLDB*, *WWW*, *ICDE*, and *ACM SIGSPATIAL GIS*. His research interests include data security and privacy, with focus on privacy-preserving transformation of microdata, private queries in location-based services, and privacy-preserving sharing of sensitive data sets. He is a member of the IEEE and the IEEE Computer Society.

Elisa Bertino is a professor of computer science at Purdue University, and serves as research director of the Center for Education and Research in Information Assurance and Security (CERIAS) and interim director of Cyber Center (Discovery Park). Previously, she was a faculty member and department head at the Department of Computer Science and Communication of the University of Milan. Her main research interests include security, privacy, digital identity management systems, database systems, distributed systems, and multimedia systems. She is a fellow of the IEEE and the ACM. She received the IEEE Computer Society Technical Achievement Award *for outstanding contributions to database systems and database security and advanced data management systems* and the IEEE Computer Society Tsutomu Kanai Award *for pioneering and innovative research contributions to secure distributed systems* in 2002 and 2005, respectively.