2015 IEEE Symposium on Security and Privacy

Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems

Yuanzhong Xu
The University of Texas at Austin
yxu@cs.utexas.edu

Weidong Cui
Microsoft Research
wdcui@microsoft.com

Marcus Peinado
Microsoft Research
marcuspe@microsoft.com

***Abstract*—The presence of large numbers of security vulnerabilities in popular feature-rich commodity operating systems has inspired a long line of work on excluding these operating systems from the trusted computing base of applications, while retaining many of their benefits. Legacy applications continue to run on the untrusted operating system, while a small hypervisor or trusted hardware prevents the operating system from accessing the applications' memory.**

In this paper, we introduce controlled-channel attacks, a new type of side-channel attack that allows an untrusted operating system to extract large amounts of sensitive information from protected applications on systems like Overshadow, InkTag or Haven. We implement the attacks on Haven and InkTag and demonstrate their power by extracting complete text documents and outlines of JPEG images from widely deployed application libraries.

Given these attacks, it is unclear if Overshadow's vision of protecting unmodified legacy applications from legacy operating systems running on off-the-shelf hardware is still tenable.

I. Introduction

The past years have seen a significant effort in the design of systems that shield applications from the operating system [19], [18], [49], [42], [27], [20], [22], [35], [10]. Such *shielding* systems typically use trusted hardware or a hypervisor to prevent the operating system from reading or writing to an application's memory and from directly tampering with its execution. The goal is to protect applications even if the operating system is adversarial.

This proposition is compelling. The never-ending stream of vulnerabilities found in large, feature-rich legacy operating systems draws into question their ability to truly protect themselves or their applications. Shielding systems promise to fill this gap by using a small, defensible trusted computing base to protect applications. In cloud hosting, shielding systems have the potential of protecting customer applications and their data from the cloud provider [39], [10].

The untrusted operating system continues to provide critical functionality, such as resource management and exposing standard interfaces. This supports important features, including multi-tasking, paging and the ability to run legacy applications with few or no modifications. Such functionality is critical for platform adoption.

However, running legacy code on an untrusted operating system is dangerous, as such code was written under the assumption that the operating system is trusted. In order to avoid breaking implicit security assumptions in the legacy code, the shielding systems has to completely insulate the application from all types of adversarial action by the untrusted operating system. This is hard, given the operating system's role in resource management and providing services to the application.

Checkoway and Shacham [17] observe that an adversarial operating system can carefully craft the return values of system calls to exploit applications. However, recent work on shielding systems [10] shows that even very complex legacy applications can be protected from such Iago attacks by drastically reducing the system call interface and by carefully checking the results of system calls before returning them to application code.

In this paper, we introduce controlled-channel attacks – a new type of side-channel attack on shielding systems. The untrusted operating system uses its control over the platform to construct powerful side channels. Shielding systems cannot rely on applications, off-the-shelf hypervisors or isolation hardware to eliminate these channels, as those components were designed for different environments. This makes it the task of the shielding system to keep side channels under control. However, existing shielding system designs ignore side channels.

In a traditional side-channel setting [47], [12], [40], [7], [8], [14], [46], [44], [52], the attacker has no control over system events such as context switches, memory accesses by other code, TLB flushes, exceptions, interrupts, page faults, and changes in mappings. Such events typically introduce high levels of noise into the channel. The attacker can eliminate the noise by averaging over many runs of the victim's code. But this has confined such attacks to a narrow set of domains, such as extracting cryptographic keys, in which the victim can be made to execute the same code over the same secret data a large number of times.

More recently, work on the Flush-Reload technique [24], [50] has shown that the amount of noise in the cache side channel can be significantly reduced if the victim and the attacker share memory. Flush-Reload has enabled several new and improved attacks [50], [30], [11], [51], [48].

The operating system's high degree of control over system events allows us to go significantly beyond existing side-channel attacks. We present a no-noise channel that permits simultaneous monitoring of large numbers of virtual addresses

DOI 10.1109/SP.2015.45

and observation of high-frequency events at high time resolution. We use this channel to construct attacks that extract large amounts of data (hundreds of kilobytes) from a single run of applications which have, until now, been beyond the reach of side-channel attacks.

Our no-noise channel uses page faults due to memory accesses by the application. We use it to extract text documents from widely used word processing tools (the FreeType font rendering engine [1] and the Hunspell spell checker [2]), to obtain outlines of JPEG images decompressed by libjpeg [3] and to undo Windows-style ASLR. In each case, a single run of the victim's code is sufficient to leak the data from the protected application. We have implemented and successfully executed these attacks on two of the most recent and sophisticated shielding systems: Haven [10] and InkTag [27].

Even though existing shielding systems typically do not include secure physical input and output, our target applications are realistic, as users can communicate securely with their applications through a cryptographically secured network connection (ssh, remote desktop). This is particularly compelling in the cloud scenario [10].

As we target legacy code, our attack model assumes that the application binaries are public. Our attacks are based on a detailed off-line analysis of the memory access patterns of these binaries. The first step of each attack is to infer the base address at which each binary is loaded. The attacker can then restrict access to particular code or data pages either by editing the page tables directly or through an interface of the shielding system. When the application tries to access one of these pages, a page fault will occur and the operating system will be invoked to resolve it. At this point, the operating system can record that the page was accessed, update the page restrictions based on the application's page-access history and resume the application. Thus, as the application executes, the operating system collects a trace of page accesses by the application. The last step of the attacks is an off-line analysis of this trace to recover the application's secret data.

One significant obstacle to our attacks lies in the fact that we can observe memory accesses only at the granularity of 4 KB pages. While x86 processors (and some shielding systems [27]) provide page-fault handlers with the full address at which the page fault occurred, we follow Haven [10], [29] and provide the operating system only with the page number (which it needs to handle the fault), but not with the 12-bit offset within the page. We design several techniques that allow the attacks to succeed in spite of being restricted by page granularity.

A second challenge lies in limiting the overhead introduced by the attacks. Page faults are expensive operations, and naïve versions of the attacks may easily increase the running time of the applications by several orders of magnitude. We describe optimization techniques that allow us to execute the attacks at modest overheads.

Many of the mitigations that have been designed for cache side-channel attacks [21], [32], [53] could, in principle, be adapted for our attacks. Application code could be rewritten to avoid memory access patterns that depend on the application's secret information. But this may impact performance and negate the goal of allowing legacy applications to run on untrusted operating systems.

Alternatively, the shielding system could try to prevent the attacks by restricting the operating system's control over page-access permissions. It could also try to detect attacks by monitoring execution time, page faults, and activity by the operating system. The challenges are to retain the operating system's ability to effectively manage resources and to avoid false positives. We discuss these options in more detail at the end of the paper.

In summary, this paper makes the following contributions:

- We introduce controlled-channel attacks as a serious threat to shielding systems, which system designers should take into account.
- We design several concrete controlled-channel attacks against widely used libraries.
- We implement these attacks efficiently on Haven and InkTag.

II. Background and Attack Model

Rather than limiting our analysis to one concrete system, we describe our attacks for a broader class of shielding systems. The goal of a shielding system is to allow legacy applications to run on legacy operating systems, but without having to trust the latter. A highly privileged monitor component constrains the operating system and prevents it from interfering with the application. The monitor is typically a hypervisor [19], [27], [20], but it can also be secure hardware [10].

The monitor protects applications even if the operating system is adversarial and actively trying to attack them. For this purpose, the monitor needs to provide a secure mechanism to initialize applications. It also needs to provide isolated execution environments for protected application execution. The former often involves cryptographically protected disk storage for application files. For the latter, the monitor has to provide memory to the application that the operating system cannot access. Hypervisor-based systems implement this by interposing on page table updates for the application by the operating system, and preventing the operating system from accessing memory allocated for the application. This technique is also used to safeguard the integrity of the layout of the application's virtual address space. The monitor also has to interpose on all context switches (e.g., interrupts) between the application and the operating system to prevent application state from leaking to the operating system.

Refinements of these techniques allow the operating system to continue performing tasks such as demand paging and scheduling in spite of the restrictions. Furthermore, the operating system provides (untrusted) services such as memory allocation or access to storage and the network to the application. As shown in [17] such interactions with the operating system may give rise to Iago attacks unless they are carefully validated.

A. Attack Model

We assume that an attacker controls the operating system. However, we leave the monitor components and any code running in the application's protected environment untouched. The class of shielding systems targeted by our attacks has the following properties:

a) Memory resource management by the operating system: The system uses virtual memory. The operating system can use demand paging to assign physical memory to a variety of applications. Thus, the operating system controls virtual to physical memory mappings in accordance with its resource management task. The shielding system may constrain the operating system in order to prevent it from reading and writing application memory and to ensure the integrity of the application's address space. For example, the operating system must not be allowed to map a particular page at an address where the application does not expect it. However, the operating system has the ability to reclaim physical pages and, thus, to remove virtual-to-physical page mappings. It must also be able to restore page mappings to handle page faults. To do so, the operating system must be able to obtain the virtual base address of the page at which the page fault occurred. We do not assume knowledge of the offset within the page.

b) Applications: The system supports largely unmodified legacy applications. Such applications typically do not take special measures to obscure their memory access patterns (with the exception of crypto code that has been hardened against cache side-channel attacks). We assume the legacy applications are public, and the attacker knows the exact versions of application binaries being targeted. In our attacks, we performed manual analysis on the source code of the applications. This is not a problem for open source software. Manual analysis on binary code is also possible, albeit more tedious.

Most of the shielding systems cited in the introduction meet these two conditions. This includes Haven and InkTag for which we have implemented our attacks. In contrast, systems like Flicker [38] or TrustVisor [37] are not included in this class, as they require significant modifications to legacy applications and, in the case of Flicker, support only a single protected region with static resources.

III. Design

In this section, we present the design of our attacks. The key intuition is to exploit the fact that a regular application usually shows different patterns in control transfers or data accesses when the sensitive data it is processing are different. We refer to them as *input-dependent control transfers* or *input-dependent data accesses*. A malicious operating system can observe input-dependent control transfers or data accesses to infer the sensitive data.

To observe input-dependent control transfers or data accesses, the operating system can induce page-fault traps by restricting access to particular code or data pages. When a function on a code page is called or when a data object on a data page is accessed, a page fault will result, and the operating system can recognize the control transfer or data access based on the page-fault address. The operating system can steal an application's secrets by repeatedly observing input-dependent control transfers and data accesses.

The main challenge in designing our attacks is that the operating system cannot observe the exact byte-granular address of a page fault but only the page's 4 KB-granular base address because a secure hypervisor or hardware can always zero out the lowest 12 bits in a page-fault address before passing it to the operating system. In fact, Intel SGX [29] already does this. When not knowing actual page-fault addresses, the operating system cannot directly detect a control transfer or a data access.

In the rest of this section, we first describe how our attacks can be launched in an ideal environment where full page-fault addresses are given. Then we describe our solutions for identifying control transfers and data accesses when page-fault addresses are only given at page-level granularity. Finally, we describe how we handle page faults in detail. We assume the base addresses of loaded modules are known and present our attack against ASLR in Section IV-D.

A. Basic Attack

```
char* WelcomeMessage( GENDER s ) {
    char *mesg;

    // GENDER is an enum of MALE and FEMALE
    if ( s == MALE ) {
        mesg = WelcomeMessageForMale();
    } else { // FEMALE
        mesg = WelcomeMessageForFemale();
    }
    return mesg;
}
```

Fig. 1: Example function with input-dependent control transfer.

```
void CountLogin( GENDER s ) {
    if ( s == MALE ) {
        gMaleCount ++;
    } else {
        gFemaleCount ++;
    }
}
```

Fig. 2: Example function with input-dependent data access.

Our attacks exploit input-dependent control transfers or data accesses to steal secrets from an application. Here we use two simple examples to explain the basic idea. This subsection explains how an attacker with access to the *complete* (byte-granular) page-fault address could proceed.

In Figure 1, we show an example function with an input-dependent control transfer. The `WelcomeMessage` function calls the `WelcomeMessageForMale` function if a user is male. It calls the `WelcomeMessageForFemale` function if a user is female. We assume the function

`WelcomeMessage` and the other two functions are on different code pages. To launch an attack, we restrict access to the code pages of `WelcomeMessageForMale` and `WelcomeMessageForFemale`. When one of them is called, a page fault will be triggered. We can tell the user's gender based on which function was called.

In Figure 2, we show an example function with input-dependent data access. The `CountLogin` function counts the number of male and female user logins by using two global variables `gMaleCount` and `gFemaleCount`. The function `CountLogin` and the two global variables are usually on different memory pages since the former is executable and the latter is writable. To launch an attack, we restrict access to the data page or pages containing `gMaleCount` and `gFemaleCount`. When one of the variables is accessed, a page fault will occur and the page-fault address will reveal the user's gender. Furthermore, we can infer the total number of male and female users by counting the number of page faults at `gMaleCount` and `gFemaleCount`.

The key to a successful attack is to recognize the input-dependent control transfers (e.g., `WelcomeMessageForMale` or `WelcomeMessageForFemale`) or the input-dependent data accesses (e.g., `gMaleCount` or `gFemaleCount`). When actual page-fault addresses are given, this can be done straightforwardly. However, when page faults are reported at page-level granularity, this becomes a challenge. Next, we present our approach to solving this problem.

B. Inferring Input-Dependent Memory Accesses

The first stage in our attacks is an offline analysis of an application's code to identify what control transfers and data accesses are input dependent and how we can use them to reveal the application's secrets. Currently, we do this manually and on a per-application basis. Here, we assume this offline analysis has identified input-dependent control transfers or data accesses and focus on how we can *infer* them when page faults are reported at page-level granularity. In other words, this section explains how we deduce that a function at a particular address has been called or a variable at a particular address has been accessed, given only a trace of page numbers, but not complete addresses.

It is quite common for a function (or data object) to share a memory page with other functions (or data objects). When a page fault happens on such a memory page, we cannot directly tell if the function (or data) of interest is being accessed. The key idea for inferring a particular function invocation or data access is to identify page-fault *sequences* that are unique to the function (or data) access.

To identify unique page-fault sequences for a specific memory access, we run the application outside the protected environment (without the shielding system) and record page-fault traces by restricting access to all pages. Upon a page fault, we record the faulting address and remove the restriction for the faulting page in order to allow application execution to proceed. Subsequently, we add the restriction again, as described in Section III-C. This gives us a trace of byte-granular page-fault addresses.

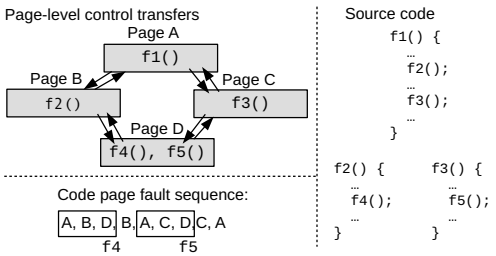


Fig. 3: The attacker can only observe page-level control transfers. However, functions sharing the same page can often be distinguished by different page-fault sequences.

We associate two addresses with each page fault: We call the address whose access triggered the page fault the *page-fault address*. We call the address of the instruction that was being executed when the page fault occurred the *instruction address* of the page fault. For a code page fault, these two addresses are identical (except for instructions that cover two memory pages and the page-fault address lies on the second page).

Control transfers: To infer a specific control transfer, we only record page faults of code pages. Let us assume we collect a set of page-fault traces $\{P_i = \{p_i^j\}\}$, where P_i represents the i-th trace, and p_i^j represents the page-fault address of the j-th page fault in the i-th trace. We collect multiple page-fault traces to have a better coverage of an application's execution paths. Since executables may be loaded at different addresses in different runs, we convert page-fault addresses to be module offsets.

For each trace P_i we generate a new trace $Q_i = \{q_i^j\}$ where q_i^j is the page base address of the j-th page fault in the i-th trace. The Q_i are of the types of traces our attacks would obtain. Let f be the target address of the control transfer we want to identify. Then for each s, t such that $p_s^t = f$, we search for the minimum $k \geq 1$ such that, for any sequence $(q_i^{j-k+1}, q_i^{j-k+2}, ..., q_i^j)$ that matches with the sequence $(q_s^{t-k+1}, q_s^{t-k+2}, ..., q_s^t)$, p_i^j equals to f. That is, for each occurrence of f in any of the P_i traces, we search for the shortest sequence of its preceding pages for which the corresponding sequence in Q_i leads only to f for all its occurrences. In general, we may find more than one such sequence for f, since different appearances of f in the traces may be preceded by different page sequences. For example, a function f may be called from several places.

Finally, we use the set of unique sequences $\{(q_s^{t-k+1}, ..., q_s^t)\}$ to identify the control transfer. In the attacks we will present in Section IV, there is usually a single page-fault sequence and the length is usually 2 or 3 for inferring a specific control transfer.

Figure 3 shows how two functions sharing the same page can be distinguished by different page-fault sequences.

Data accesses: To infer a data access at a specific memory

address, we record full byte-granular page-fault traces of both code and data pages without the shielding system. We first identify all the data page faults due to accesses to the specific memory address. Then, for each of these data page faults, we search for a minimum sequence of *code* page faults (right before the data page fault) that can be used to uniquely infer the data access. We combine all the identified sequences to derive a set of page-fault sequences for inferring the given data access. As before, there is usually a single page fault sequence and the length is usually 2 or 3 for inferring a data access of interest.

We might not necessarily know the addresses of some data before the application runs; they can be dynamically allocated. But we do know the instructions that access them (from our manual offline analysis), and thus can still identify accesses to them using the same technique as above. This often leaks enough information for us to extract fine-grained application data.

Discussion: The algorithm described so far assumes there is a single thread. For multi-threaded applications, we simply record a thread identifier with each page fault and regroup the traces based on thread ids. This allows us to treat the page faults for each thread as a separate trace and to apply the same algorithm.

In practice, to find a short page-fault sequence, we may need to do multiple iterations between the manual offline analysis for identifying useful memory accesses and the programmatic analysis for identifying page-fault sequences.

The quality of the identified page-fault sequences depends on the execution coverage of an application. We can improve their quality and the analysis accuracy by using different inputs to drive the application's execution for better code coverage. How exactly this can be done and how much improvement can be achieved is beyond the scope of this paper. Our focus is to demonstrate the feasibility of controlled-channel attacks.

C. Handling Page Faults

We just described how we can identify a set of page-fault sequences for inferring a control transfer or data access. In this section, we describe how we handle page faults in detail.

Code pages: While tracking all code pages of the target process would give us the most information, it would also make the attack slow. Instead, we only track a small set of relevant pages, which we refer to as *tracking pages*. The basic approach works as follows. First, we include all pages in the page-fault sequences identified in the offline analysis as the set of tracking pages, and restrict access to them when an application starts. Second, when a page fault happens, we log the page-fault event, enable access to the page, and remove access to the previous page. This basic approach is straightforward and works in general. But we need to improve it to handle the following issues.

When we only track a subset of the pages, we may see false positives for the unique page-fault sequences described in Section III-B. Recall that we computed these sequences from traces in which we tracked *all* pages. For example, let (p_a, p_c) be such a sequence. The sequence of page accesses (p_a, p_b, p_c) does not match this sequence. However, if we do not track page p_b, we will observe (incorrectly) an access sequence (p_a, p_c). To avoid such false positives, we reduce a full page fault trace to a trace that we would have observed when tracking the set of selected pages. This reduction is done by first removing unmonitored pages and then merging consecutive identical pages in the trace. Then we search for false positive sequences in the reduced trace. For every false positive sequence like $(p_a, p_b, ..., p_c)$, we add the page p_b into the set of tracking pages.

An instruction may cross two consecutive code pages. If both pages are in the set of tracking pages, the application will hang when we use the basic approach to handle this instruction since we will see alternating page faults on these pages. To deal with this problem, we first identify if we have an instruction cross two contiguous tracking pages. If so, when we see consecutive page faults on these two pages, we make both accessible. When the next page fault occurs, we remove access to both pages.

Data pages: For some attacks, we need to track data page faults. We often do not need to track them throughout the entire attack but during a particular function's execution. In such a case, we start the tracking of data pages when detecting a function's invocation (based on a particular code page-fault sequence) and stop it when the function finishes (based on another code page-fault sequence).

When a data page fault happens, we do not always remove access to previous data pages. For instance, we do not need to do it if we only care about whether a specific data access *occurs* in a function. If we also care about the *number* of data accesses in a function, we need to remove access to previous data pages in order to incur repeated data page faults. In the latter case, we need to handle data page faults in a way similar to code page faults to allow the application to make progress.

An x86-64 instruction may access up to two memory locations. For example, the instruction `call [funcptr]` reads the function target from the `funcptr` and writes the return address to the stack. Another example is the instruction `movs` which copies data from one memory location to another. In the worst cases, if both memory accesses are cross two memory pages, we may need to handle four data page faults and enable access to all of them to let the instruction execute. Our solution for this problem has two parts. First, for consecutive data page faults on contiguous pages, we make both accessible. Second, if we observe alternating page faults of two data pages, we make both accessible. When identifying alternating patterns, we conceptually consider two contiguous pages as a single page.

IV. Attacks

In this section, we first demonstrate how we can launch our attacks to extract fine-grained data from three widely used applications, FreeType [1], Hunspell [2], and libjpeg [3]. Then we describe how we attack Address Space Layout

Randomization (ASLR) [13] to reveal the base addresses of loaded modules.

A. FreeType

FreeType is a user-level font library that renders text onto bitmaps. It is widely used in a variety of software products, including Linux distributions, the Android and iOS platforms, Ghostscript, and OpenJDK. It supports different font formats, including TrueType, the most common format for fonts on Microsoft Windows and Mac OS. We describe an attack is on TrueType. Other font formats are subject to similar attacks.

In TrueType fonts, a glyph for a character is represented as a collection of line and curve commands as well as a collection of hints. FreeType executes the commands and processes the hints to draw a glyph onto a bitmap. Since different glyphs have different commands and hints, the control flows for rendering them are different. Therefore, these control flows are dependent on the character that is being rendered. We exploit this to infer the rendered text. We do not need to induce data page faults in this attack.

The render function for TrueType is `TT_Load_Glyph`. This function is invoked for rendering every character with its glyph. We first identify two sets of page-fault sequences for inferring the start and end of this function. After that, a naive approach would be to track all accesses to all code pages during each invocation of this function and to use them to infer each character. This would work but impose a significant performance overhead. Instead, we identify a small subset of code pages subject to the constraint that the number of page faults over an invocation of `TT_Load_Glyph` uniquely identifies the character being rendered.

Our attack starts with an offline analysis in which we render all distinct characters (i.e., letters and punctuation marks) and find the page-fault counts for the selected set of code pages that can uniquely identify each character. During the online attack, we use the two sets of page-fault sequences to identify the start and the end of `TT_Load_Glyph` and log the page-fault counts of the selected code pages for each invocation of `TT_Load_Glyph`. Finally, we deterministically identify each rendered character by comparing these counts with the counts we obtained from the offline analysis. The last stage can be done either online or offline.

B. Hunspell

Hunspell is a popular spell checking tool widely used in many software packages, including Mac OS X and Google Chrome. Hunspell loads words in a dictionary into a hash table in memory and checks if a word is in the hash table to decide the correctness of its spelling. The hash table uses separate chaining with linked lists to handle hash collisions. In other words, the hash table starts with an array of pointers to linked lists. The indices to the array are hash values. Each linked list contains all words with the same hash value. Figure 4 shows an example of the hash table.

When inserting a word into the hash table, Hunspell accesses multiple data pages, including the page of the pointer

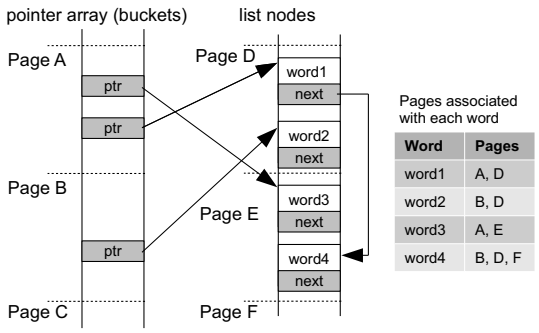


Fig. 4: The hash table in Hunspell.

to the linked list and the pages of the nodes on the linked list. Similarly, when looking up a word from the hash table, Hunspell accesses the same data pages in the same order as accessed during insertion. If we know which sequence of data pages is being accessed when a word is being inserted, we can tell when it is being checked by observing the same sequence of data pages accessed during lookup. We assume knowledge of the dictionary used by Hunspell. Since Hunspell inserts dictionary words sequentially, we know the order of insertion as well.

In Hunspell, the function `HashMgr::add_word` does insertion and the function `HashMgr::lookup` does lookup. We identify four sets of page fault sequences for inferring the start and the end of `HashMgr::add_word` and `HashMgr::lookup`.

During the online attack, we use the four sets of page-fault sequences to infer the invocations of `HashMgr::add_word` and `HashMgr::lookup`. During each invocation, we trap all data-page accesses. Given the recorded page-fault sequence, we first identify the sequence of data pages accessed for each word in the dictionary. Then we use these sequences to infer the words looked up by Hunspell. This can be done either online or offline. Note that at any data page fault, we do not make other pages inaccessible; removing access to pages is only done at code page faults.

It is possible that Hunspell accesses the same sequence of data pages when inserting two different words into the hash table. When this happens, there is more than one choice for the word being checked, resulting in an ambiguity. Fortunately, our experiments show that the degree of ambiguity is low, even though we only have page-level access traces. This is due to the *low correlation* between a word's hash value and its location in the input dictionary. Since words are inserted sequentially according to their order in the dictionary, the list nodes of words adjacent in the dictionary are likely to reside on the same page or contiguous pages; on the other hand, adjacent words in the dictionary typically have very different hash values such that the pointers to their linked lists are on different pages. As a result, the possibility that multiple words share exactly the same sequence of data pages is relatively low. To mitigate the ambiguity, we leverage a language model to identify which word is more likely to appear than others. This

helps us eliminate most ambiguities.

In most cases, we can detect if a word does not appear in the dictionary by observing a data-page access in the hash table that is not followed by accesses to data pages storing list nodes. However, if the word has the same hash as one or more words that are in the dictionary, we will see the same page-fault sequence as for the word corresponding to the last node in the linked list for that hash value. In these rare cases, our attack will return that word.

Hunspell supports affixes. It tries to remove an affix from a word and look it up again if the original word is not found. This means that we may see multiple lookups for a single word in an input document. To avoid showing multiple words with different affixes for a single input word, we also track the invocations of the function `Hunspell::spell` in our attack, which is invoked exactly once for each word in the document. Multiple invocations to `HashMgr::lookup` during a single `Hunspell::spell` invocation indicate different trials for the same input word, and we only show the word in the last lookup because previous ones might have failed.

C. JPEG

JPEG is a commonly used lossy image compression standard. Libjpeg is a JPEG codec implemented by the Independent JPEG Group. The library itself or a direct derivative of it is used in countless applications for JPEG encoding and decoding.

Given an original bitmap image, the JPEG encoder first divides it into blocks of 8×8 pixels. It then performs a *discrete cosine transform* for each block. The output of the transform is a coefficient matrix sized 8×8, which is passed to *quantization* and finally *compression*. Quantization is the only lossy operation in the whole process. To decode a JPEG image, the decoder *decompresses* and *dequantizes* the encoded blocks to get the recovered coefficient matrices for them. Then, for each coefficient matrix, it performs an *inverse cosine transform (IDCT)*, which outputs the 8×8 bitmap block. By combining all decoded blocks, the decoder can generate the entire bitmap image.

Libjpeg has several IDCT implementations. All of them have the same basic structure. They have two `for` loops, iterating over the 8 columns and the 8 rows in the coefficient matrix respectively, as IDCT is a two-dimensional operation. The default IDCT function used in libjpeg is `jpeg_idct_islow`. A code snippet of it is shown in Figure 5. Our attack focuses on `jpeg_idct_islow` and is applicable to other IDCT functions.

The IDCT function is small enough to fit on a single code page, and it does not invoke other functions. Thus, there are no input-dependent control transfers we can leverage. Instead, we exploit input-dependent data accesses caused by a performance optimization in the IDCT functions. The normal processing for a row/column in IDCT requires heavy computation. However, if all but the first element in a row/column are zeroes, the row/column is constant and the computation is fairly simple. The IDCT functions in libjpeg first check if this condition is true and, if so, do the simplified computation. For a row or column, the number of data-page accesses in the standard (complex) version of the computation is much larger than for the simplified version. If we actively restrict access to data pages during execution of the IDCT function, we observe a significant difference in the data page fault counts for the two code paths. We try to keep only a single data page accessible at a time, by removing access to other data pages when handling a data page fault, unless an alternating pattern is detected indicating that an instruction needs to access two data pages.

```
GLOBAL(void) jpeg_idct_islow (j_decompress_ptr cinfo,
  jpeg_component_info * compptr, JCOEFPTR coef_block,
  JSAMPARRAY output_buf, JDIMENSION output_col)
{
  ...
  /* Pass 1: process columns from input... */
  inptr = coef_block;
  quantptr = (ISLOW_MULT_TYPE *) compptr->dct_table;
  wsptr = workspace;
  for (ctr = DCTSIZE; ctr > 0; ctr--) {
    /* Due to quantization, we will usually find that
     * many of the input coefficients are zero,
     * especially the AC terms.  We can exploit this
     * by short-circuiting the IDCT calculation for any
     * column in which all the AC terms are zero. In
     * that case each output is equal to the DC
     * coefficient (with scale factor as needed). With
     * typical images and quantization tables, half or
     * more of the column DCT calculations can be
     * simplified this way.
     */
    if (inptr[DCTSIZE*1]==0 && inptr[DCTSIZE*2]==0 &&
          inptr[DCTSIZE*3]==0 && inptr[DCTSIZE*4]==0 &&
          inptr[DCTSIZE*5]==0 && inptr[DCTSIZE*6]==0 &&
          inptr[DCTSIZE*7]==0) {
      /* AC terms all zero */
      ... SIMPLE COMPUTATION ...
      inptr++; quantptr++; wsptr++;
      continue;
    }
    ... COMPLEX COMPUTATION ...
    inptr++; quantptr++; wsptr++;
  }
  /* Pass 2: process rows from work array... */
  wsptr = workspace;
  for (ctr = 0; ctr < DCTSIZE; ctr++) {
    if (wsptr[1]==0 && wsptr[2]==0 && wsptr[3]==0
        && wsptr[4]==0 && wsptr[5]==0 && wsptr[6]==0
        && wsptr[7]==0) {
      /* AC terms all zero */
      ... SIMPLE COMPUTATION ...
      wsptr += DCTSIZE;
      continue;
    }
    ... COMPLEX COMPUTATION ...
    wsptr += DCTSIZE;
  }
}
```

Fig. 5: IDCT function in libjpeg.

The number of page faults during an IDCT invocation reveals the number of constant rows and columns in the block, which corresponds to the relative complexity of the block. Our attack exploits this fact to recover an image in which each pixel corresponds to a block in the original image. For a gray scale image, each pixel has one color component. We compute the value for a pixel in the recovered image by normalizing the number of data page faults in `jpeg_idct_islow` for its corresponding block in the original image. Our normalization is done based on the maximum/minimum number of data page faults in `jpeg_idct_islow` for all blocks. The recovered

image is often similar to the result of edge detection, because edge blocks tend to be more complex than others.

In color images, a pixel has multiple color components. The meaning of each color component is defined by the format of the image, which is also called a *color space*. These color components are encoded separately; the IDCT function is invoked multiple times for each block to decode all color components in a fixed order. Assuming the color space is known, we can compute the value of each component separately for a pixel in the recovered image, based on the number of page faults during the corresponding `jpeg_idct_islow`. Our normalization is done separately for each color component over all blocks.

However, we do not know what color space is used in the JPEG image. But fortunately, there are only a few commonly used ones. For example, the most common color space used in JPEG is YCbCr, which has three color components: luminance (Y), chromatic blue (Cb) and chromatic red (Cr); another common color space is RGB, used in high-quality JPEG images, which also has three components (red, green, blue). We simply try to recover the image multiple times assuming different color spaces, and manually select the one with the best visual effects.

To count the number of data page faults in `jpeg_idct_islow`, we identify two sets of page-fault sequences for inferring the start and the end of this function. By counting the number of invocations of `jpeg_idct_islow`, we know the total number of blocks. To recognize the dimension of an image, we need to know the number of rows or columns. To do so, we identify one set of page-fault sequences for inferring the start of the function `decompress_onepass` since it is invoked for each row of blocks in the image. The number of invocations of `decompress_onepass` tells us the number of rows of blocks in the image.

D. ASLR

So far we have assumed we know the base addresses of loaded executables (e.g., `.exe` and `.dll` files) in the description of our attacks. However, this will not be the case if a shielding system implements Address Space Layout Randomization (ASLR) to randomize the bases of executables in memory. Existing shielding systems like Haven and InkTag have not implemented ASLR for protecting applications. But we anticipate the support of ASLR in future shielding systems due to the concern of return-oriented programming (ROP) attacks. Next we describe our attack against ASLR.

The key idea for our ASLR attack is that we can leverage the very first few page faults on each executable to distinguish them. In modern OSes like Windows and Linux, the execution of an executable always starts from a predefined entry point. The exception is the loader itself (e.g., `ntdll.dll` on Windows). The loader does not have an entry point like regular executables. But the execution of a loader always starts from a deterministic location as well (e.g., `LdrInitializeThunk` in `ntdll.dll`), which is also the very first code executed in user mode for a process.

To launch our ASLR attack, we first perform an offline analysis to identify the very first few page faults for each executable that can be used to distinguish it from other executables loaded in an application. During the online attack, we assume all the executables are scattered in a large contiguous memory range. This is a weaker assumption than assuming the knowledge of individual memory ranges for loaded executables since we could leverage the sizes of memory ranges in the latter case. When a process starts, we restrict access to all mapped memory ranges. Then the first code page fault in the process is on the loader. We use it to identify the base address of the loader and enable access to all its pages. Then the next code page fault will be on a different executable. We use the next few consecutive code page faults to identify the executable and enable access to all its pages. We repeat this process until we identify all loaded executables. We do not necessarily need to identify all but the relevant executables in our ASLR attack.

An executable of interest may be loaded by delayed loading. To identify the location of a dynamically loaded executable, we restrict access to newly mapped memory regions and track initial code page faults on them. Then we use the first few consecutive code page faults on the new memory regions to identify the dynamically loaded executable.

V. Implementation

In this section, we first present our implementation of controlled-channel attacks on two shielding systems, Haven [10] and InkTag [27]. We have implemented our attacks on prototypes of InkTag and Haven which the authors of these systems have kindly made available to us. After that, we describe how we realize attacks against FreeType, Hunspell, and libjpeg on both Haven and InkTag. Finally, we present the implementation of our ASLR attack on Windows since Haven and InkTag do not support ASLR.

A. Implementation of controlled-channel attacks on Haven

Haven relies on trusted hardware (SGX [29]) rather than a hypervisor to constrain the operating system and provide isolated execution environments (enclaves) for applications. The only Haven software running outside an enclave is an untrusted driver – concerned primarily with memory management and with interacting with SGX – and an untrusted user-mode runtime that provides an interface between the trusted code and the operating system.

The code inside a Haven enclave consists of a Windows application, the Drawbridge library operating system [41] and a *Shield* module. Drawbridge, which consists of Windows user-mode libraries and a user-mode kernel, provides the application with a full Windows interface, but has only a very narrow set of dependencies on the underlying system. This set of dependencies is significantly smaller and simpler than the system call interface of Windows or Linux. The Shield module protects the remaining dependencies from adversarial actions

by the host operating system. For example, the Shield module implements an encrypted and integrity-protected file system, limiting storage interactions with the untrusted host operating system to reading and writing of crypto-protected disk blocks.

SGX includes side-channel protections that keep the CPU performance counters from being used to construct side channels against enclaves [29]. These protections appear adequate for the small-TCB security applications for which SGX was designed [26], as the developers of security applications can be expected to mitigate higher-level side channels in software. Our attack succeeds in the context of Haven because most legacy software is not hardened against side channels.

Our attack relies on the following aspects of the Haven prototype: First, SGX leaves all page tables under the control of the host operating system and implements a new, independent memory protection mechanism. A memory access will fail if it is disallowed under either of the two mechanisms. Thus, our attack code has unrestricted access to the page tables, and, by editing them appropriately, it can force code running inside enclaves to cause page faults.

Second, a page fault during enclave execution results in SGX transferring control to the regular operating-system page-fault handler specified in the interrupt descriptor table (IDT) (after saving and scrubbing the CPU register context). SGX also zeros out the bottom twelve bits of the faulting address, revealing only the page number of the faulting address to the operating system, but not the offset within the page.

Third, the Shield module has to call the operating system to map and unmap memory regions in the enclave and to change their page-access permissions. This provides the attacker with a precise and up-to-date map of the enclave's memory layout. Individual Windows binaries (e.g., FreeType) are easily recognized as a sequence of consecutive regions of specific length with specific read, write and execute permissions—corresponding to the different sections of the binary. Other mapped regions such as the heap can be identified by similar heuristics. When adding knowledge about the (highly deterministic) order in which Drawbridge loads binaries and allocates memory, these heuristics become quite reliable.

The calls to map memory regions inform us not only about the loading addresses of binaries, but also about the moment at which they are loaded. We can use this as a trigger for starting the attack. That is, we can let Drawbridge boot unencumbered, monitoring only requests to map and unmap enclave memory at next to no overhead. When we observe that a binary of interest (e.g., FreeType) has been loaded, we activate the tracing of page accesses.

The Shield module's memory mapping calls are not easily avoided. SGX limits enclave code to run in user mode, while the instructions to modify enclave memory permissions can only be executed in kernel mode.

We have implemented the attack by modifying the Haven driver, leaving the trusted components of Haven (Shield module and Drawbridge) unchanged. We inserted our code in the driver's handler functions for memory mapping calls by the Shield module. The code implements the heuristics described above to detect the loading of any of the modules targeted by our attacks on FreeType, Hunspell and libjpeg. When one of the modules is detected, we execute the corresponding attack as described in Section IV.

For efficiency reasons, we implement access restrictions by directly editing the page tables, rather than calling Windows functions. Setting a reserved bit (e.g., bit 51) in the x86-64 page table entries causes all accesses to the corresponding pages to result in a page fault that can be easily recognized by our page-fault handler. This method is preferable to using the present bit, as it minimizes interference with the Windows memory manager.

We avoid page table shootdowns by affinitizing the processes of the target applications to a single core. This has no noticeable effect on performance for our three single threaded target applications. For multithreaded applications, one can envision adversarial operating systems whose schedulers can choose different tradeoffs between parallelism, the cost of frequent page faults and the completeness of the page-fault trace.

We install our page-fault handler by overwriting the corresponding address in the IDT. This ensures that we can resolve the page faults triggered by our attacks without unnecessarily executing a large amount of operating system code. This implementation is critical to minimizing our overhead, as our attacks trigger a large number of page faults. After determining that the fault was due to our attacks, the page-fault handler invokes an attack-specific procedure that resolves and logs the page fault and adjusts the access restrictions as described in Section IV. All three attacks against FreeType, Hunspell and libjpeg added 1972 lines of C code and 156 lines of assembly code to the Haven driver.

We also implemented the attacks for the variant of Haven used in the performance evaluation of [10]. This version of the prototype does not use the Haven driver and relies on Windows system calls for memory management. For this version, we compiled the attack code we had added to the Haven driver into a separate stand-alone driver. We also added 144 lines of C code to the untrusted Haven runtime to notify our driver every time trusted code requests a memory management operation (allocation, freeing, change of page permissions) from the operating system. Our evaluation in the next section is based on this version of the implementation. Lacking information about the performance characteristics of a potential future implementation of SGX, we did not attempt to model the impact of SGX on the performance of our attacks.

B. Implementation of controlled-channel attacks on InkTag

InkTag uses a trusted hypervisor to protect applications running on an untrusted operating system. A protected application runs inside a *high-assurance process* (HAP). The hypervisor protects the secrecy and integrity of a HAP's memory page contents by encryption and hashing. Memory management is still performed by the untrusted operating system. To ensure that every memory page is mapped at the virtual address requested by the application, InkTag does not

allow the untrusted operating system to directly modify the page tables of a HAP. Instead, to update the page tables, the untrusted operating system must make a hypercall into the hypervisor via the paravirtualized function `set_pte`. The hypervisor then verifies the request and updates the page tables for the untrusted operating system.

The implementation of our attacks on InkTag relies on the following. First, InkTag allows the untrusted operating system to map and unmap memory pages to perform paging. We exploit this fact to restrict access to a memory page by pretending to page it out (i.e., clearing the present bit in its page table entry while keeping it in memory).

Second, InkTag lets the untrusted operating system handle page faults for a HAP. Currently the hypervisor clears all general-purpose CPU registers before passing control to the operating system during a page-fault interrupt. Unlike SGX, the hypervisor provides the operating system with the full faulting address. To make our attacks more robust, we only consume the page base address of a page fault.

Third, InkTag requires all the code used by a HAP to be compiled into a static binary. It also loads the static binary at a fixed virtual address. This makes our attack easier since we do not need to deal with ASLR.

We implemented our controlled-channel attacks on InkTag by modifying the x86-64 page-fault handler in Linux. To restrict access to a memory page, we simply clear the present bit in its page table entry without actually swapping the page out of memory. We track the set of pages manipulated by our attack. When a page fault happens, we first check if it is a manipulated page. If so, we handle it based on the algorithm presented in Section III-C. If not, we pass it to the existing page-fault handler.

All three attacks against FreeType, Hunspell, and libjpeg added 799 lines of C code to the Linux kernel running on InkTag.

C. Implementation of Attacks against Applications

We realized controlled-channel attacks against FreeType, Hunspell, and libjpeg on Haven and InkTag. For several reasons, we could not use the off-the-shelf binaries. In the case of Haven, we did not find official Windows binaries for the latest versions of the three open source libraries and had to compile the source code ourselves. InkTag only supports Linux console applications and requires an application to be compiled into a *single* static binary. That made it impossible to run official Linux binaries of the three libraries on InkTag. Instead, we again had to build our own binaries from source code.

Application	Version
FreeType	2.5.3
Hunspell	1.3.3
libjpeg	9a

Fig. 6: Application versions used in the controlled-channel attacks on both InkTag and Haven.

Figure 6 shows the versions of the three open source applications we targeted in our attacks. We used the same versions for our attacks on Haven and InkTag. For Haven, we compiled the open source code by using Microsoft Visual Studio's C/C++ compiler (version 18.00.30501) [4] and used the system binaries shipped with Windows (e.g., `ntdll.dll` and `kernel32.dll`). For InkTag, we used the GCC compiler (version 4.4.5) in the uClibc tool chain [6].

Our attacks are sensitive to code optimization because it may eliminate some control transfers that we could otherwise leverage for inference. For instance, a function call disappears if the callee is inlined into the caller. We may not be able to track a function call if the caller and the callee are on the same memory page, which a compiler frequently does for performance optimization. To make our attacks as realistic as possible, we used the same compiler options as other binaries on Windows and Linux. For Windows binaries running on Haven, we used the full optimization (i.e., `/Ox`) and inlining (i.e., `/Ob2`) options of the Microsoft Visual Studio C/C++ compiler. For Linux binaries running on InkTag, we used the level-2 optimization (i.e., `-O2`) in the GCC compiler, where possible targets for inlining include specified inline functions, static functions that are only called once (i.e., `-finline-functions-called-once`), and small functions (i.e., `-finline-small-functions`).

We built simple console applications to drive the libraries. The FreeType source package does not have a command-line application. We wrote a simple command-line application that calls into the FreeType library for each letter in an input file to render it onto a bitmap buffer with the Times New Roman font, a popular one in the TrueType font format. The Hunspell source package includes a command-line application that does spell checking on an input file. We used it together with the en_US dictionary in the source package. For libjpeg, we wrote a simple command-line application that calls into the libjpeg library to decode a JPEG image and saves the result into a BMP file.

D. Implementation of the ASLR attack on Windows

For evaluation purposes, we implemented our ASLR attack on Windows since Haven and InkTag do not support ASLR. We use Windows kernel APIs (e.g., `PsSetCreateProcessNotifyRoutineEx`) to track process creation and identify the target process when it is created. Then we use Windows kernel APIs (e.g., `PsSetLoadImageNotifyRoutine`) to track memory regions allocated for executables in the target process. We use the same approach as described in Section V-A to directly manipulate page tables. We restrict access to memory pages by setting the Non-Executable bit (bit 63). This allows us to avoid unnecessary page faults on data pages. The prototype of our ASLR attack was implemented as a kernel driver. It has 1644 lines of C code and 156 lines of assembly code. Part of the code is shared with our implementation on Haven.

group size	Haven words	Haven %	InkTag words	InkTag %
1	46864	75.16	48864	78.37
2	9964	15.98	9372	15.03
3	3546	5.69	2880	4.62
4	1100	1.76	852	1.37
5	485	0.78	275	0.44
6	222	0.36	60	0.10
7	49	0.08	14	0.02
8	48	0.08	16	0.03
9	45	0.07	0	0.00
10	30	0.05	20	0.03

Fig. 7: Distribution of words in the en_US Hunspell dictionary: More than 75% of the words can be uniquely identified by the attack. For more than 95% there are at most three choices.

accuracy of recovery		Haven words	Haven %	InkTag words	InkTag %
recovered original word	no ambiguity	25320	63.75	27179	68.43
	rec. 2-group	6042	15.21	5751	14.48
	rec. 3-group	1985	5.00	2554	6.43
	rec. $\geq$4-group	2869	7.22	890	2.24
recovered without affix	no ambiguity	1974	4.97	2291	5.77
	rec. 2-group	602	1.52	460	1.16
	rec. 3-group	213	0.54	145	0.37
	rec. $\geq$4-group	291	0.73	186	0.47
not recovered		423	1.06	263	0.66

Fig. 8: Accuracy of the Hunspell attack on *The Wizard of Oz* before applying the language model: over 63% of the words were recovered exactly.

	Haven words	Haven %	InkTag words	InkTag %
recovered exactly	35273	88.81	35760	90.03
recovered without affix	2880	7.25	2896	7.29
not recovered or incorrectly resolved ambiguity	1566	3.94	1063	2.68

Fig. 9: Accuracy of the Hunspell attack on *The Wizard of Oz* after applying the language model: over 88% of the words were recovered exactly.

VI. Evaluation

In this section, we demonstrate that our attacks are effective by showing that they can extract text and image data from protected Inktag high-assurance processes and Haven enclaves, as well as inferring the base addresses of relevant binaries in an ASLR-protected Windows process. We also show that the impact of our attacks on the applications' execution times is generally moderate.

We ran the Haven experiments on a Lenovo IdeaCentre H530s with a quadcore Intel i5-4440 Haswell processor running at 3.1 GHz, 6 GB of RAM and a 256 GB Samsung 840 PRO SSD. The machine was running Windows 8.1 Pro. Before starting the experiments we copied the application binaries and their support files as well as the test inputs for the evaluation to Haven's encrypted virtual hard disk.

The Inktag experiments were run on a Dell OptiPlex 980 with a quadcore Intel i7-860 processor running at 2.8 GHz, 8 GB of RAM and a 160 GB Western Digital WD1600HLFS SSD. The machine runs Linux 2.6.36, on which InkTag is built.

A. Effectiveness of the attacks

1) FreeType: We used an ASCII version of the book *The Wonderful Wizard of Oz* [9] (downloaded from [5]) as the test input to our FreeType application described in Section V. The size of this input file is 213,087 bytes. It contains 39,719 words.

During the run of the FreeType application, our attack code in the operating system produced a page-fault trace. Our post-processing tool recovered an ASCII file from the trace. We repeated the experiment ten times on both Haven and InkTag and compared the ten output files of the attack with the input file. All ten recovered files were identical to the input file.

2) Hunspell: We ran our Hunspell application from Section V over the same input file we used in the FreeType evaluation. Again, we collected a page-fault trace of the run and used our attack specific post-processing tool to recover an ASCII file from it. We repeated the experiment ten times.

Unlike the FreeType attack, the Hunspell attack does not recover the input file exactly. Multiple dictionary words mapping to the same set of pages, removed affixes and missing words in the Hunspell dictionary can cause the attack output to differ from the original input. Furthermore, the attack does not recover punctuation marks and white spaces.

Figure 7 shows statistics on the effectiveness of the attack. It shows how many words from the en_US Hunspell dictionary have a unique page-fault pattern (group size 1). These words are uniquely identified by the attack. The figure also shows how many words share their page-fault pattern with n other words. For such words, the attack can only recover a group of $n+1$ words which includes the word that is being looked up.

More than three quarters of the words from the dictionary are uniquely identified by their page-fault pattern during the hash table lookup. More than 95% of the words are in groups of at most three words (group size ≤ 3). No word is in a group of more than 10 words. The small differences between the statistics for InkTag and Haven are due to differences in the memory allocation code of each system.

To resolve ambiguities caused by multiple dictionary words being mapped to the same set of pages, we used a commercial English language model. This model contains 130,840 unigrams, 1,573,498 bigrams and 1,239,511 trigrams. Given multiple candidates, we first check if there is a matching trigram and, if so, pick the one with the highest probability. If there is no matching trigram, we check the bigrams. If there is a matching bigram, we pick the one with the highest probability. Otherwise, we check the unigrams. If there is a matching unigram, we pick the one with the highest probability. Otherwise, we keep all the candidate words.

Figure 8 and Figure 9 show the accuracy of our Hunspell attack for *The Wonderful Wizard of Oz* before and after applying the English language model. More than 63% of the

```
Folklore, legends, myths and fairy tales
have followed childhood through the ages,
for every healthy youngster has a wholesome
and instinctive love for stories fantastic,
marvelous and manifestly unreal. The winged
fairies of Grimm and Andersen have brought
more happiness to childish hearts than all
other human creations.
```

```
folklore *legend* myths and fairy *tale*
have *follow* childhood through the *age*
for every healthy youngster has a wholesome
and instinctive love for [store] fantastic
marvelous and *manifest* unreal the [wine]
*fairy* of [grill] and Andersen have brought
more happiness to childish *heart* than all
other human *create*
```

Fig. 10: A sample output of the Hunspell attack on Haven (right) and the original input (left). Brackets ([ ]) denote words that the attack could not uniquely identify and for which the language model failed to resolve the ambiguity correctly. Asterisks (*) denote words with missing prefixes or suffixes. The sample text is the first paragraph of the Introduction in *The Wonderful Wizard of Oz*.

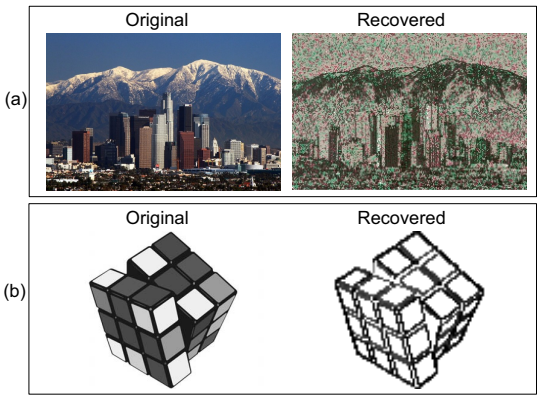


Fig. 11: A small sample of the images we used to test the libjpeg attack.

words were recovered exactly without the language model. The accuracy was improved to 88% after using the language model to resolve ambiguities. If we include words recovered without affix, the accuracy reaches 96%. Less than 1.1 percent of the words were not recovered at all because they were either not in the dictionary (e.g., names) or skipped by Hunspell (e.g., numbers).

Overall, our Hunspell attack demonstrates significant information leakage that permits recovery of almost the entire input text. Even without punctuations, the output of our attack tends to be easily comprehensible. Figure 10 shows a sample.

3) Libjpeg: We downloaded a test set of 18 JPEG images from various sites on the internet. We tried to collect a diverse set of images, including complex high-resolution photos as well as simpler logo-style images.

We ran the libjpeg application from Section V on these input files. For each run, our attack code collected a page-fault trace. Our post-processing tool extracted a BMP file from each of the traces.

Figure 11 displays two examples of pairs of inputs and images recovered by the attack. Figure 15 in the appendix displays all other image pairs in our test sample. The quality and accuracy of the extracted outputs varies depending on the input image. However, in most cases, enough information was leaked to easily identify important features of the image.

4) ASLR: We ran our ASLR attack against FreeType, Hunspell, and libjpeg on Windows. For every loaded executable, we need to capture only the first two code page faults on the executable to recognize it. For each application, we ran the experiments 10 times. In all experiments, we correctly identified all loaded executables.

B. Performance

This section analyzes the overhead introduced by the attacks. The goal is to analyze whether the attacks cause delays in the execution of the applications that are so large as to draw attention to them. On normal commodity systems, events such as interrupts, network and disk activity, virus scans and periodic activity by different system services introduce jitter into the execution time of applications. Multi-user cloud hosting environments, which are the target of recent shielding systems [10], [39], display an even higher level of background noise due to network delays, activity by other users on the same physical machine or virtual machine migration. The question we try to answer in this section is whether the delays caused by the attacks could plausibly be hidden in this background noise.

1) FreeType: Figure 12 compares the baseline running time of the FreeType application with the running time when under attack (attack time). The numbers under the *whole file* column are averaged over ten runs over the entire 208 KB Wizard of Oz input file. The overhead is 3.74x on Haven and 32.1x on InkTag, with total attack running times of 19.3 seconds on Haven and 280.21 seconds on InkTag. These times are not insignificant. However, the FreeType application is effectively a microbenchmark. It renders hundreds of thousands of characters into a memory buffer in a tight loop. Real applications typically would intersperse font rendering with other operations such as waiting for keyboard input or, if running in the cloud, communicating screen contents over the network to a remote terminal.

We consider rendering an entire screen full of characters to the user as an example of an expensive font rendering operation a typical application might perform. To approximate this operation, we have run the attack on chunks of 5 KB from the original input file. The 5 KB columns in Figure 12 display the averages over ten runs over the first ten non-overlapping 5 KB chunks from the input file. The running times are 0.52 seconds on Haven and 6.62 seconds on InkTag. The running time on Haven appears small enough to plausibly disappear in the timing noise of cloud and even local systems.

The page-fault count (pf count) on Haven is around half the page-fault count on InkTag. This difference is the result

	whole file		5 KB	
	Haven	InkTag	Haven	InkTag
baseline time (s)	5.16	8.72	0.18	0.21
attack time (s)	19.3	280.21	0.52	6.62
overhead	3.74x	32.1x	2.89x	31.7x
pf count (million)	28.90	52.97	0.69	1.27
time per pf (ns)	489.5	5125.2	491.0	5054.7
post-proc. time (s)	< 100	< 100	< 10	< 10

Fig. 12: Performance of the FreeType attack.

	whole file		last chapter
	Haven	InkTag	InkTag
baseline time (s)	0.12	0.12	0.051
attack time (s)	2.94	11.95	0.089
overhead	25.2x	99.6x	1.75x
pf count (million)	5.84	2.41	0.0085
time per pf (ns)	484.2	4955.6	4517.2
post-proc. time (s)	< 20	< 10	< 5

Fig. 13: Performance of the Hunspell attack.

	562 KB image		36 KB image	
	Haven	InkTag	Haven	InkTag
baseline time (s)	0.08	0.12	0.04	0.014
attack time (s)	16.77	42.59	0.50	2.84
overhead	209.6x	354.9x	12.5x	202.8x
pf count (million)	35.8	8.97	0.95	0.56
time per pf (ns)	482.7	4735.5	466.0	5035.2
post-proc. time (s)	< 5	< 5	< 5	< 5

Fig. 14: Performance of the attack on libjpeg on two test JPEG images. The first is a 1920×1282, 562 KB, YCbCr color space image (Figure 11-a); the second is a 800×600, 36 KB gray scale image (Figure 11-b).

of radically different binaries (due to different compiler optimization strategies) and a less optimized implementation of the attack on InkTag. The page-fault handling time is significantly lower on Haven as a result of the highly optimized low-level page-fault handling code we were able to use with Haven. On InkTag, the restrictions of the hypervisor interface result in longer page-fault handling times. In particular, every edit of a page table entry requires a separate hypercall at the cost of a virtual machine exit. This causes the attack to have a higher overhead on InkTag.

The post-processing times appear unproblematic, as the attacker can do all post-processing off-line on separate machines.

2) Hunspell: We have done a similar analysis for the Hunspell attack (Figure 13). Here, the running time of the attack when spell checking the entire 208 KB input file is 2.94 seconds on Haven and 11.95 seconds on InkTag, resulting in overheads of 25.2x on Haven and 99.6x on InkTag.

Again, the running time on InkTag is noticeable, but somewhat less so on Haven. Also, the input file (an entire book) appears to be significantly larger than an average document. In general, reducing the document size beyond a certain point will have a limited effect, as a constant part of the overhead is incurred while Hunspell reads in the dictionary.

On InkTag, we have used the following optimization to avoid this overhead. We have observed the `malloc` behavior in InkTag's C library (`uClibc`) to be deterministic during dictionary loading. As Hunspell itself is also deterministic, the layouts of the dictionary data structures in memory are identical in every run. This allows us to record them offline and to omit tracing the insertion of the dictionary words. For small documents, this reduces the overhead significantly. For example, checking the last chapter of the book (77 words) only requires 0.089s.

While the attacker does not know a priori how long the document is, the attack code can track the overhead incurred and interrupt the attack if the overhead exceeds a certain threshold.

Again, due to different binaries and different variants of the attack implementation, the numbers of page faults on InkTag and on Haven differ. As in the case of FreeType, the page-fault handling time for the InkTag attack is significantly larger. The post-processing time does not appear to be an obstacle.

3) Libjpeg: In Figure 14, we report the performance of decoding the two images shown in Figure 11 using libjpeg. Figure 11-a is a 562 KB YCbCr color image with 1920×1282 pixels; Figure 11-b is a 36 KB gray scale image with 800×600 pixels.

For Figure 14-a, the running time is 16.77 seconds (209.6x) on Haven and 42.59 seconds (354.9x) on InkTag; for Figure 14-b, the running time is 0.5 seconds on Haven (12.5x) and 2.84 seconds (202.8x) on InkTag. The overhead is due to the fact that we are adding 50 to 300 data page faults to each invocation of one of the most performance-sensitive functions in libjpeg. The high cost of handling page faults and editing page tables on InkTag also contributes significantly to the overhead, though it incurs fewer page faults compared to Haven. The post-processing time is only a few seconds.

4) ASLR: Since we incur only two code page faults for each loaded executable, the performance overhead of our ASLR attack is negligible.

VII. Discussion and Mitigations

This paper shows that controlled-channel attacks are a real threat to shielding systems that has, so far, not been taken into consideration. In addition to the page-access-based attacks demonstrated in this paper, there are several other potential information channels controlled-channel attacks could exploit: thread scheduling, patterns in the application's system calls to the operating system or low-noise cache side channels constructed by the operating system. The threats posed by these channels remain to be explored.

Like traditional cache side-channel attacks, the attacks in this paper exploit memory access patterns that depend on application secrets. Existing mitigation strategies for cache side-channel attacks could, in principle, be adapted for the attacks in this paper. Such mitigations can be applied at the application level [21] or at the system level [32], [53].

At the application level, the main strategy is to rewrite the application such that its memory access pattern does not depend on sensitive data. This can be done manually [15]

or with the help of a special compiler [21]. This approach is bound to impact application performance and may require a significant development effort. The task is furthermore complicated by the fact that, in general, sensitive application information (e.g., JPEG images) may be larger and more distributed than the crypto key targets of typical cache side-channel attacks. Even identifying the sensitive data completely may require an exhaustive security analysis of the application.

System-level mitigations appear to pose fewer obstacles. At one extreme, the shielding system could prohibit paging by the operating system. But this would also disable an important feature of shielding systems. A less invasive solution would be to prohibit paging only for a smaller subset of the application's pages. For example, keeping the operating system from paging executable pages in application binaries would prevent all attacks described in this paper. More generally, given an analysis of its data dependent memory access patterns, the application might choose the set of pages that has to be protected and use an InkTag-style mechanism to communicate this set to the shielding system.

Self-paging [25] as suggested in [10] moves paging from the operating system into the application. The operating system continues to manage *how much* memory each application controls. Enabling self-paging for shielding systems would require new hardware (in the case of Haven) or new paging interfaces in hypervisor-based shielding systems. It would also require significant changes to legacy operating systems and additional self-paging code in the protected processes.

Hiding the application's memory access pattern through noise injection or ORAM techniques [36] is another direction. Similarly, one could attempt to obscure the layout of the binaries in memory through variants of fine-grained ASLR (e.g., [23]). However, the tradeoffs between the resulting protection and the associated overhead are unclear and would require further research.

Finally, the application or the shielding system could attempt to detect artifacts of the attack. Page fault counts and execution time could be used as signals. As analyzed in Section VI, the execution time may not be a reliable indicator for an attack. The attacker could also respond by interrupting or aborting the attack once it incurs a certain overhead or page fault count. The trade offs between effectiveness and potential for false positives of this approach remain to be explored.

VIII. Related work

A. Removing the operating system from the TCB

Feature rich commodity operating systems are complex and have been plagued by a long list of security vulnerabilities. This has inspired work on removing the operating system from the trusted computing base (TCB). More recently, enabling users to run applications on cloud hosting services without having to trust the service provider has emerged as a second powerful motivation.

One line of work tries to minimize the software TCB and targets small, secure applications or application fragments that are especially written for a small-TCB environment [38], [37], [35], [45]. The absence of legacy allows for the design of simple new interfaces between the secure applications and the rest of the system. At the same time, lack of support for legacy applications and a low level of system support for new applications are serious obstacles to platform adoption.

Thus, a second line of work seeks to leave existing applications and platforms as intact as possible, while excluding the operating system from the TCB. These shielding systems support legacy applications and most features of legacy operating systems. Overshadow [19], [42], CHAOS [18], InkTag [27], SP3 [49] and AppShield [20], use a hypervisor to isolate application memory and CPU state from the operating system. Virtual Ghost [22] combines compiler instrumentation and runtime checks on operating system code to achieve similar isolation goals. Haven [10] relies on Intel SGX [29] CPU extensions to protect applications. In these systems, the operating system manages resources (e.g., memory, files) for protected applications.

B. Attacks against protected applications and defenses

Iago attacks [17] demonstrate that isolating memory and CPU state is not sufficient to protect legacy applications on shielding systems. Applications interact with the operating system via the system call interface. Legacy applications generally do not check return values for possible attacks by the operating system. The operating system can exploit this lack of verification and return carefully crafted values, such as new memory mappings that overlap with the stack, to corrupt application behavior.

InkTag [27] and Virtual Ghost [22] interpose a layer of trusted code between the application and the complex system call interface. The code performs additional checks for a number of system calls in order to defeat the known Iago attacks. Haven [10] uses a library operating system to substantially reduce the size and complexity of the system call interface, making protection from Iago attacks a more tractable task.

In contrast to Iago attacks, controlled-channel attacks extract application data through side channels without changing application behavior or returning adversarial values to system calls.

C. Side-channel attacks

There is a large body of work on various types of side channels, including power [34], timing [33], [16], process memory footprints [31], network packet sequence numbers [43] and resource usage statistics [54]. Cache side-channel attacks [47], [12], [40], [7], [8], [14], [46], [44], [52] are most closely related to the attacks described in this paper, as both exploit secret-dependent memory accesses by the victim.

Realistic attack models involve unprivileged attackers who have to cope with high levels of platform noise. Typically, the attacker gathers data over many runs and eliminates the noise by statistical techniques [16], [12]. The need to observe the victim over many runs turns crypto code with a fixed key into the target of most attacks. Hund et al. attack kernel-level

ASLR [28], again using repeated observation and statistics to overcome the noise.

If physical memory is shared between the victim and the attacker, even an unprivileged attacker has direct control over cache lines used by the victim. Such settings enable the much more efficient Flush-Reload-based cache side channels [24], [50]. Several authors demonstrate efficient crypto attacks [50], [30], [11], [48]. Zhang et al. [51] describe three interesting non-crypto attacks.

A number of differences separate this line of work from the channel and the attacks explored in this paper. Our attack targets a different class of system (shielding systems). This results in different adversary models (untrusted OS vs. unprivileged VM or user mode code), different channels (page faults vs. cache misses), different challenges (page-fault granularity vs. false cache hits/misses) and different techniques (page-fault sequences vs. precise polling). Our adversary is more powerful, but our attack is also able to extract much richer information (full text and images).

The page-fault channel, which is not available to unprivileged attackers, allows us to receive event notifications deterministically (through a page fault). This enables robust tracking of high-frequency events even under system loads that would add significant noise into the cache channel.

Going beyond the control flow technique of [51], two of our three attacks rely critically on tracking data flow (e.g., tracking the hash table in Hunspell). It is unclear how the large number of virtual addresses that is required could be tracked with Flush-Reload in practice. Furthermore, the required data pages are unlikely to be available to the attacker of Flush-Reload, as they would not be shared under standard Flush-Reload models (PaaS, memory deduplication).

IX. Conclusions

This paper introduces controlled-channel attacks as a new type of attack against shielding systems and demonstrates that controlled-channel attacks are a real threat that must be addressed in the design of those systems.

We design specific controlled-channel attacks against three widely used application libraries and implement and execute them on two of the most modern and sophisticated shielding systems. Our attacks are able to extract entire documents and approximate versions of JPEG images from protected processes on these systems. We also discuss a number of potential mitigations.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful feedback. We are very grateful to Andrew Baumann for his help with Haven and to Youngjin Kwon, Michael Z. Lee and Emmett Witchel for their help with InkTag.

References

[1] Freetype. http://www.freetype.org/.
[2] Hunspell. http://hunspell.sourceforge.net/.
[3] libjpeg. http://libjpeg.sourceforge.net/.
[4] Microsoft Visual Studio. http://msdn.microsoft.com/en-us/vstudio.
[5] Project Gutenberg. http://www.gutenberg.org/.
[6] uClibc. http://www.uclibc.org/.
[7] Onur Acıiçmez and Çetin Kaya Koç. Trace-driven cache attacks on AES. Cryptology ePrint Archive, Report 2006/138, 2006.
[8] Onur Aciiçmez, Werner Schindler, and Çetin K Koç. Cache based remote timing attack on the AES. In *Topics in Cryptology–CT-RSA 2007*, pages 271–286. Springer, 2007.
[9] L Frank Baum. *The wonderful wizard of Oz*. George M. Hill Company, May 1900.
[10] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.
[11] Naomi Benger, Joop van de Pol, Nigel Smart, and Yuval Yarom. "Ooh aah...just a little bit": A small amount of side channel can go a long way. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2014.
[12] Daniel J. Bernstein. Cache-timing attacks on AES. Available at: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005.
[13] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: an effcient approach to combat a broad range of memory error exploits. In *USENIX Security Symposium*, 2003.
[14] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. 2006.
[15] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. IACR ePrint Archive, Report 2006/052, 2006.
[16] David Brumley and Dan Boneh. Remote timing attacks are practical. In *USENIX Security Symposium*, 2003.
[17] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.
[18] Haibo Chen, Fengzhe Zhang, Cheng Chen, Ziye Yang, Rong Chen, Binyu Zang, Pen-chung Yew, and Wenbo Mao. Tamper-resistant execution in an untrusted operating system using a virtual machine monitor. Technical Report FDUPPITR-2007-0801, Parallel Processing Institute, Fudan University, August 2007.
[19] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.
[20] Yueqiang Cheng, Xuhua Ding, and R Deng. AppShield: Protecting Applications Against Untrusted Operating System. Technical Report SMU-SIS-13, Singapore Management University: School of Information Systems, November 2013.
[21] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, pages 45–60, 2009.
[22] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual Ghost: Protecting Applications from Hostile Operating Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.
[23] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium*, 2012.
[24] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – bringing access-based cache attacks on AES to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, pages 490 –505, May 2011.
[25] Steven M. Hand. Self-paging in the Nemesis operating system. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 73–86, 1999.
[26] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Carlos Rozas, Vinay Phegade, and Juan del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *The Second Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.
[27] Owen S. Hofmann, Alan M. Dunn, Sangman Kim, Michael Z. Lee, and Emmett Witchel. InkTag: Secure applications on an untrusted operating system. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[28] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *IEEE Symposium on Security and Privacy*, 2013.

[29] Intel Corp. *Software Guard Extensions Programming Reference*, September 2013. Ref. #329298-001 http://software.intel.com/sites/default/files/329298-001.pdf.

[30] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, cross-vm attack on AES. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2014.

[31] Suman Jana and Vitaly Shmatikov. Memento: Learning secrets from process footprints. In *IEEE Symposium on Security and Privacy*, 2012.

[32] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. In *USENIX Security Symposium*, 2012.

[33] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology*, pages 104–113, 1996.

[34] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, pages 388–397, 1999.

[35] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. MiniBox: A two-way sandbox for x86 native code. In *USENIX Annual Technical Conference (ATC)*, 2014.

[36] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanović, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[37] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy*, 2010.

[38] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An Execution Infrastructure for Tcb Minimization. In *ACM European Conference in Computer Systems (EuroSys)*, 2008.

[39] Emmanuel Owusu, Jorge Guajardo, Jonathan McCune, Jim Newsome, Adrian Perrig, and Amit Vasudevan. OASIS: On achieving a sanctuary for integrity and secrecy on untrusted platforms. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[40] Colin Percival. Cache missing for fun and profit. In *BSDCan 2005*, Ottawa, 2005.

[41] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library OS from the top down. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[42] Dan RK Ports and Tal Garfinkel. Towards application security on untrusted operating systems. In *USENIX Workshop on Hot Topics in Security (HotSec)*, 2008.

[43] Zhiyun Qian, Z Morley Mao, and Yinglian Xie. Collaborative TCP sequence number inference attack: how to crack sequence number under a second. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[44] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[45] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using ARM TrustZone to build a trusted language runtime for mobile applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[46] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(2):37–71, 2010.

[47] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, and Maki Shigeri. Cryptanalysis of DES implemented on computers with cache. In *Proceedings of the 2003 Cryptographic Hardware and Embedded Systems*, pages 62–76, 2003.

[48] Joop van de Pol, Nigel Smart, and Yuval Yarom. Just a little bit more. In *CT-RSA*, 2015.

[49] Jisoo Yang and Kang G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *VEE*, 2008.

[50] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium*, 2014.

[51] Yinqian Zhang, Ari Juels, Michael Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

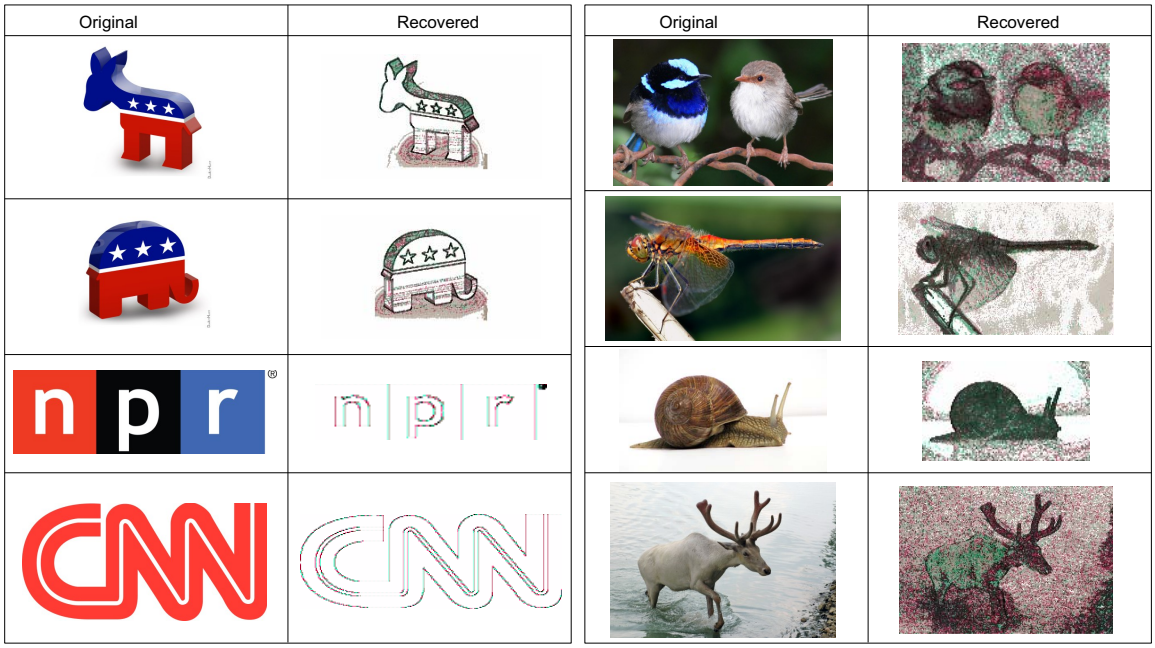
[52] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[53] Yinqian Zhang and Michael K Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[54] Xiaoyong Zhou, Soteris Demetriou, Dongjing He, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, Carl A Gunter, and Klara Nahrstedt. Identity, location, disease and more: Inferring your secrets from android public resources. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

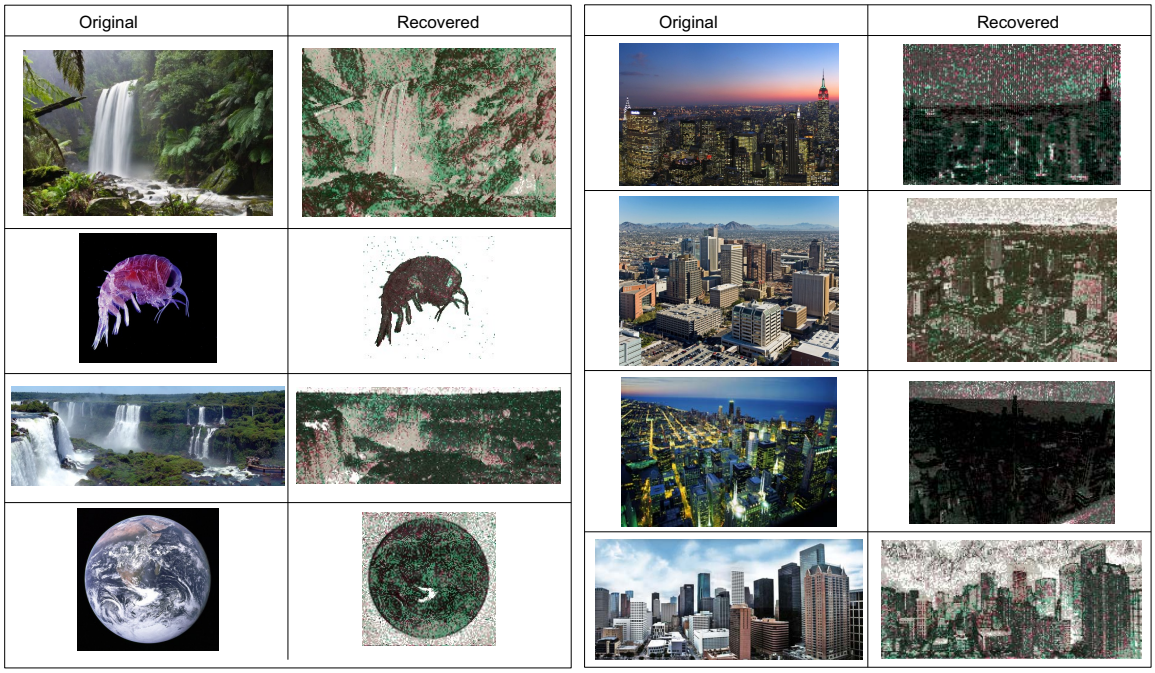
APPENDIX

We ran our libjpeg attack on a set of JPEG images, including logos, animals, nature-related images and cities, downloaded from Bing Images and Wikipedia. The results are shown in Figure 15.



(a) From Bing Images

(b) From Wikipedia Animal page



(c) From Wikipedia Nature page

(d) From Wikipedia City page

Fig. 15: Test set of images for the libjpeg attack.