Tappan Zee (North) Bridge: Mining Memory Accesses for Introspection

Brendan Dolan-Gavitt
Georgia Tech
brendan@cc.gatech.edu

Tim Leek
MIT Lincoln Laboratory
tleek@ll.mit.edu

Josh Hodosh
MIT Lincoln Laboratory
josh.hodosh@ll.mit.edu

Wenke Lee
Georgia Tech
wenke@cc.gatech.edu

ABSTRACT

The ability to introspect into the behavior of software at runtime is crucial for many security-related tasks, such as virtual machine-based intrusion detection and low-artifact malware analysis. Although some progress has been made in this task by automatically creating programs that can passively retrieve kernel-level information, two key challenges remain. First, it is currently difficult to extract useful information from user-level applications, such as web browsers. Second, discovering points within the OS and applications to hook for *active monitoring* is still an entirely manual process. In this paper we propose a set of techniques to mine the memory accesses made by an operating system and its applications to locate useful places to deploy active monitoring, which we call *tap points*. We demonstrate the efficacy of our techniques by finding tap points for useful introspection tasks such as finding SSL keys and monitoring web browser activity on five different operating systems (Windows 7, Linux, FreeBSD, Minix and Haiku) and two processor architectures (ARM and x86).[1]

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Design, Security

Keywords

Introspection; active monitoring; reverse engineering

[1]This work is sponsored by the Assistant Secretary of Defense for Research & Engineering under Air Force Contract #FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

http://dx.doi.org/10.1145/2508859.2516697 .

1. INTRODUCTION

Many security applications have a need to inspect the internal workings of software. Host-based intrusion detection systems, malware analyses, and digital forensics all depend to some degree on being able to obtain information about software that is by design undocumented and hidden from public view. Thus, to operate correctly, security software is typically built on *reverse engineering*, the art and practice of elucidating the undocumented principles on which software is built.

Unfortunately, reverse engineering is expensive, time consuming, and requires a high degree of expertise. The problem is exacerbated by the fact that, to protect against tampering, security applications are often hosted in environments separated from the target being inspected, such as a separate virtual machine. Because of this, their visibility into the target is often limited to low-level features such as memory and CPU state, and any higher-level information must be reconstructed based on reverse engineered knowledge.

This problem, which we will refer to as the *introspection problem*, has been approached by a number of recent research efforts such as Virtuoso [11] and VMST [12]. Existing systems, however, have a number of limitations. First, they focus on retrieving kernel-level information. However, a great deal of security-relevant information exists only at user-level, such as URLs being visited by the browser, instant messages and emails sent by desktop clients, and system and application log messages. Second, they require that the desired information be accessible through some public interface (a public API in the case of Virtuoso, and a userland program or kernel module in the case of VMST). This means that some security-relevant information may be inaccessible to such tools. Finally, Payne et al. [25] argue that many security applications need some form of *active monitoring*; that is, they need to be notified when certain system events occur. Current solutions to the introspection problem provide no way of locating places in the system where it would be useful to interpose.

In this paper, we attempt to address the limitations of past solutions by examining a rich source of information about system and application activity: memory accesses observed at runtime. Our key insight is that a memory accesses made at different points in a program can be treated as streams of related information. For example, when visiting a URL, a web browser must write to memory the URL that is be-

ing visited, and it will generally do so at the same point in the program. By intercepting memory accesses made at this program point we can observe all URLs visited. These program points, which we call *tap points*, provide a natural place to interpose to extract security-relevant information, and could be integrated into an active monitoring system such as Lares [25].

There are several challenges that must be overcome to make use of tap points. The first is the sheer amount of data that must be sifted through. In ten minutes' worth of execution on a Windows 7 system, for example, we observed a total of 18.9 *million* unique tap points which read and wrote a total of 32.8 gigabytes of data. To overcome this challenge, we make use of techniques from information retrieval and machine learning, described in Section 4, to quickly zero in on the tap points that read or write information relevant to introspection.

Second, simply setting up an environment in which one can observe every memory access made by the whole system (OS and applications) poses a challenge. Whole-system emulators such as QEMU [4] provide the necessary basis for such instrumentation, but intercepting and analyzing every memory access online is not practical: the resulting system is so slow that network connections time out and the guest OS may think that programs have become unresponsive. To solve this problem, we add *record and replay* to QEMU, which allows executions to be recorded with low overhead. Our heavyweight analyses are then run on the replayed execution to analyze every memory access made *without* perturbing the system under inspection. We describe our system, Tappan Zee Bridge (TZB),[2] in detail in Section 5.

Finally, previous systems have required significant effort to support new architectures. This problem has become more pressing in recent years, as ARM-based devices such as smartphones have exploded in popularity. Because TZB looks at memory accesses, rather than inspecting binary code, it naturally supports a wide variety of architectures with minimal effort. To demonstrate this, our evaluation includes the ARM architecture in addition to x86, and the techniques we describe easily generalize to other architectures.

The remainder of this paper is structured as follows. Section 2 precisely defines what a tap point is. We then explore that definition and its impact on the scope of our work and the assumptions it rests on in Section 3. Section 4 describes techniques for finding tap points of interest. We then discuss our system, Tappan Zee Bridge (TZB), which implements these techniques, as well as PANDA (Platform for Architecture-Neutral Dynamic Analysis), a new dynamic analysis platform on which TZB is built, in Section 5. We evaluate TZB in Section 6, and show that it is capable of finding tap points useful for introspection in a wide variety of applications, operating systems, and architectures. Finally, we describe the limitations of our approach in Section 7, related work in this area in Section 8, and offer concluding remarks in Section 9.[3]

[2]So named because the northbridge on Intel architectures traditionally carried data between the CPU and RAM.

[3]All software described in this paper (i.e. PANDA and TZB) is open source and can be downloaded at `http://github.com/moyix/panda/`.

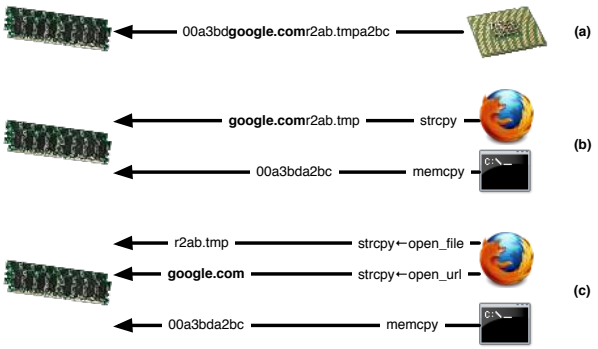

Figure 1: Three different ways of defining a tap point: (a) as a single stream of information from the CPU to RAM ; (b) split up according to program and location within program ; (c) split up according to program, location within program, and calling context.

2. DEFINING TAP POINTS

At the heart of our approach is an abstraction on top of memory accesses made by the CPU, the *tap point*. A tap point is a point in a system at which we wish to capture a series of memory accesses for introspection purposes; however, the exact definition of "a point in a system" will make a great deal of difference in how effective our approach can be.

A naive approach to defining tap points would be to simply group memory accesses by the program counter that made them (e.g., EIP/RIP on x86 and R15 on ARM). This approach fails in two common cases: first, memory accesses made by bulk copy functions, such as `memcpy` and `strcpy`, would all be grouped together, which would commingle data from different parts of the program into the same tap point. In addition, looking only at the program counter would conflate accesses from different programs.

Instead, we define tap points as the triple

$$(caller, program\_counter, address\_space)$$

Including the caller and the address space (the `CR3` register on x86, and the `CP15 c2` register on ARM) separates out memory accesses into streams that should, in general contain the same type of data.[4] Figure 1 shows the effect of choosing various definitions of a tap point when looking for the place where the browser writes the URL entered by the user ("google.com"). At the coarsest granularity (a), one can simply look at all writes from the CPU to RAM; however, the desired information is buried among reams of irrelevant data. Separating out tap points by program and program counter (b) is better, but still combines uses of `strcpy` that contain different information — in this case, a filename and a URL. By including the calling context (c), we can finally obtain a tap point that contains just the desired information.

[4]Making use of tap points defined this way in the real world is slightly more difficult, since a program's address space will differ and its code may be relocated by ASLR. These complications can be overcome with a minor amount of engineering, however.

It is possible that some tap points may require deeper information about the calling context (for example, if an application has its own wrapper around `memcpy`), but in practice we have found that just one level of calling context is usually sufficient. In addition, because TZB uses a whole-system emulator that can watch every call and return, we can obtain the call stack to an arbitrary depth for any tap point. This makes it easy to add extra context for a given tap point, if it is found that doing so separates out the desired information. Examples of tap points that require more than one level of callstack information are given in Sections 6.1.2 and 6.1.3.

Conversely, one might wonder whether this definition of a tap point may split up data that should logically be kept together. To mitigate this problem case, we introduce the idea of *correlated tap points*: we can run a pass over the recorded execution that notices when two tap points write to adjacent locations in memory in a short period of time (currently 5 memory accesses). The idea is that these tap points may be more usefully considered jointly; for example, a single data structure may have its fields set by successive writes. These writes would come from different program counters, and hence would be split into different tap points, but it may be more useful to examine the data structure as a whole. By noticing this correlation we can analyze the data from the combined tap point.

3. SCOPE AND ASSUMPTIONS

The goal of Tappan Zee Bridge is to find points at which to interpose for active monitoring. More precisely, our goal is to speed the current entirely manual process by which applications or operating systems are reverse engineered in order to locate tap points for active monitoring. It should be noted that we do not aim to surpass those manual efforts. We have no automatic way, for instance, of knowing for certain if a tap point will fail to output crucial data or, alternately, spew out superfluous information under some future conditions. This is a separate problem to which we see no ready solution. Static analysis of candidate tap points or extensive testing are good stop-gaps, but nothing short of fully understanding enormous binary code bases can really give complete assurance that a tap point won't miss or cause false alarms in the future.

In this section, we explore how our definition of a tap point and our focus on active monitoring shape the scope of our work.

First and most obviously, our focus on memory accesses necessarily limits our scope to information that is read from or written to RAM at some point. Although this is quite broad, there are notable exceptions. For example, TRESOR [24] performs AES encryption without storing the key or encryption states in RAM by making clever use of the x86 debug registers and the AES-NI instruction set. Aside from such special cases, however, this assumption is not particularly limiting.

Second, our goal of finding tap points suitable for active monitoring motivates a design that treats memory accesses at tap points as sources of *streaming* data. Our algorithms, therefore, typically work in a streaming fashion as the system executes, remembering only a fixed amount of state for each tap point. Although this is a natural fit for active monitoring, where events should be reported as soon as possible,

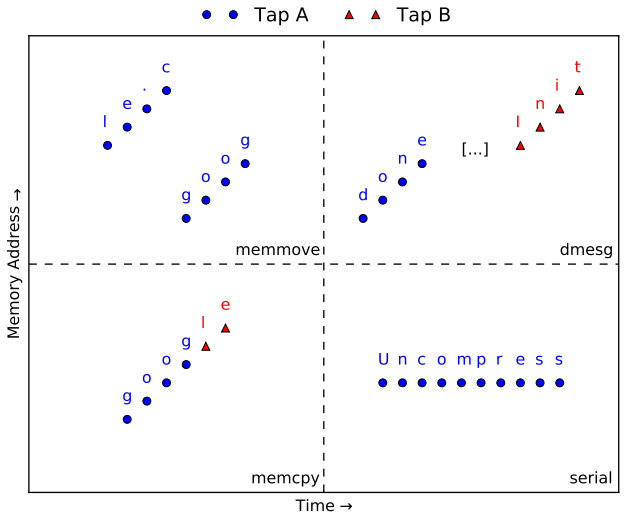

Figure 2: Patterns of memory access that we might wish to monitor using TZB.

it makes handling data whose *spatial* order in memory differs from its *temporal* order as it is accessed more difficult.

Third, the *encoding* of the data sought must be to some extent guessable. For example, to search for a string, one must know what encodings are likely to be used by the system to represent strings. In general this is not a severe limitation, but it does come up; we discuss one such case in Section 6.2.2.

Finally, the use of calling context in the definition of a tap point raises the question of how much context is necessary or useful. Our current system uses only the most recent caller, but we have seen both situations where this is not enough and where it is too much. Overall, however, one level of calling context has proved to be a reasonable choice for a wide variety of introspection tasks.

To better illustrate the boundaries of our technique, consider Figure 2, which plots the address of data written by different tap points over time for four patterns of memory access. In the bottom two quadrants, we have cases that are challenging, but currently well-supported by TZB. In the bottom-left, a standard `memcpy` implementation on x86 makes a copy in 4-byte chunks using `rep movsd`, and then does a two-byte `movsw` to get the remainder of the string. Because the access occurs across two different instructions, TZB sees two different tap points. Our tap point correlation mechanism correctly deduces that the accesses are related, however, because they operate on adjacent ranges in a short span of time.

The case shown in the bottom right quadrant would be tricky if we looked only at memory access spatially and not temporally. Here, a utility function writes data out to a serial port by making one-byte writes to a memory-mapped I/O address.[5] Because TZB sees these memory writes in temporal order, ignoring the address, the data is seen normally and the analyses we describe all operate correctly.

[5]Although not reported in this paper, this case is one we actually encountered while experimenting with an embedded firmware.

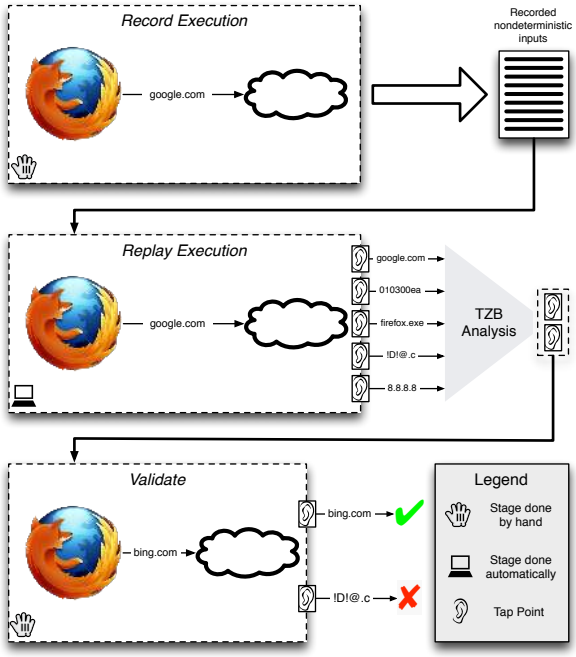

Figure 3: The workflow for using TZB to locate points at which to interpose for active monitoring.

The upper quadrants show cases that are currently not handled by TZB. In the upper left, `memmove` copies a buffer in reverse order when the source and destination overlap. Thus, when viewed in temporal order, a copy of a string like "12345678" would be seen by TZB as "56781234". This case is unlikely to be handled by TZB without a significant redesign, as its view of memory accesses is inherently streaming.

Finally, the upper right, which represents the case of `dmesg` on Linux, is an example of the "dilemma of context". Although the function, `do_syslog`, that writes log data to memory is called from multiple places (creating multiple tap points), it writes to the same contiguous buffer. Unlike the `memcpy` case, a significant amount of time may pass before the next function calls `do_syslog`, and so our tap correlation, which only considers memory accesses within a fixed time window, will not notice that the tap points ought to be grouped together. We believe that this case could be overcome with additional engineering work, but this is left to future work.

4. SEARCH STRATEGIES

To find useful *tap points* in a system—places from which to extract data for introspection—using Tappan Zee Bridge, one begins by creating a recording that captures the desired OS or application behavior. For example, if the end goal is to be notified each time a user loads a new URL in Firefox, one would create a recording of Firefox visiting several URLs. This recording is made by emulating the OS and application inside of the dynamic analysis platform PANDA (described in more detail in Section 5.1), which can capture and record all sources of non-determinism with low overhead, allowing for later deterministic replay. Next, one can run one or more analyses that seek out the desired information among all memory accesses seen during the execution. Analyses in TZB take the form of PANDA plugins that are called on each memory access made during a replayed execution and, at the end, write out a report on the tap points analyzed. Finally, the tap points found should be validated to ensure that they do, in fact, provide the desired information. Such assurance can be gained either by examining the data in the tap point in new executions, or by examining the code around the tap point. This workflow is illustrated in Figure 3.

In this section, we describe three different ways of finding tap points grouped according to a standard epistemic classification scheme [26]: searching for "known knowns"—tap points where the content of the desired data is known; searching for "known unknowns"—tap points where the kind of data sought is known, but its precise format is not; and finally "unknown unknowns"—tap points where the type and format of the data sought are not known, and we are instead simply trying to find "interesting" tap points.

4.1 Known Knowns

The simplest case is finding data that one knows is likely to be read or written by a tap point, and where the encoding of the data is easily guessed. For example, to find a tap point that can be used to notify the hypervisor whenever a URL is entered in a browser, one can visit a known sequence of URLs, and then monitor all tap points, searching for specific byte sequences that make up those URLs. The same holds for other data whose representation when written to memory is predictable: filenames, window titles, registry key names, and so on. For this kind of data, simple string searching is usually sufficient to zero in on the few tap points that handle the data of interest, and in our experience it is one of the most effective techniques for finding useful tap points.

4.2 Known Unknowns

A second tap point application involves finding tap points for things about which we have limited knowledge. We can easily assemble corpora of exemplars to represent a semantic class: English prose, kernel messages, or mail headers. These examples need not come from tap points but can easily be collected directly from interacting with the operating system itself. From such a corpus, we can readily build a statistical model, with which we can build a distance measure for scoring and ranking tap points by how close their contents are to the model.

In addition to such statistical methods, we can also search using an oracle. This is the case, for example, with tap points that write encryption keys. Although the exact key may not be known in advance, we can check whether a given byte string is a valid decryption key by trying to decrypt our sample data.

4.3 Unknown Unknowns

The final strategy for finding useful tap points is also the least focused. If there is no specific introspection quantity sought, one might instead wish to find interesting tap points, for some suitable definition of "interesting." To support this scenario, TZB offers a form of unsupervised learning—clustering—to group together tap points that handle similar data. The idea is that one can then examine exemplars from each cluster, rather than being forced to look through a large

number of tap points. Thus, our use of clustering functions as a form of *data triage*.

5. IMPLEMENTATION

In this section, we describe both the dynamic analysis platform employed to build TZB, but also TZB-specific algorithmic and data-structure solutions.

5.1 PANDA

TZB makes extensive use of the Platform for Architecture-Neutral Dynamic Analysis (PANDA), which was developed by the authors in collaboration with Northeastern University.

PANDA is based upon version 1.0.1 of the QEMU machine emulator [4]. QEMU is an excellent and common choice for whole-system dynamic analysis for two main reasons. First, performance is good (about 5x slowdown over native). Second, every basic block of guest code is disassembled by the host in order to emulate, which means that there are opportunities to interpose analyses at the basic block or even instruction level, if desired. QEMU lowers instructions to an intermediate language (IL) in order to employ a single back-end code generator, the Tiny Code Generator (TCG). This IL means dynamic analyses can potentially be written once and re-used for all 14 architectures supported by QEMU. Further, this version of QEMU is capable of booting and running modern operating systems such as Windows 7 (earlier versions of QEMU such as 0.9.1 cannot).

There are three main aspects to PANDA that make it very convenient for building dynamic analyses. First, PANDA provides a plug-in architecture that readily permits writing guest analyses in C and C++. Plug-in code is executed from a number of standard callback locations: before and after basic blocks, memory read and writes, etc. This is not unlike the schemes employed in other whole-system dynamic analysis platforms such as BitBlaze [29] and S2E [7]. In addition, plugins can export functionality that can then be used in other plugins, allowing complex behavior to be built up from simple components. From a software engineering perspective, PANDA's plugin architecture allows the various analyses supported by TZB to be cleanly separated from the main emulator, which makes for a much more comprehensible and maintainable codebase.

The second aspect of PANDA that makes it an excellent dynamic analysis platform is nondeterministic record and replay (RR). In our formulation of RR, we begin a recording by invoking QEMU's built-in snapshot capability. Subsequently, we record all inputs to the CPU, including `in`s, interrupts, and DMA. Recording imposes a small overhead (10-20%) but not enough to perturb execution. During replay, we revert to a snapshot and proceed to pull CPU inputs from a log when required. Unlike many other RR schemes, we do not record and replay device inputs, which means we cannot "go live" at any point during replay. But we can perform repeated replays of an entire operating system under arbitrary instrumentation load without worrying about this perturbing application or operating system operation. This capability is vital to TZB: without record and replay, the heavyweight analyses we perform would make the system unusably slow.

The final aspect of PANDA worth mentioning is its integration of LLVM. QEMU lowers basic blocks of guest code to its own IL, which PANDA can, additionally, re-render as basic blocks of LLVM code via a module extracted from S2E. We omit further discussion of this capability as it is not used by TZB.

5.2 Callstack Monitoring

As explained in Section 2, tap points need information about the calling context. Keeping track of this information requires some knowledge about the CPU architecture on which the OS is running, and so we decided to encapsulate this task into a single plugin. TZB's other analyses can then query the current call stack to arbitrary depth by invoking `get_callers` and not worry about the details described in this section.

To track call stack information, the `callstack` plugin examines each basic block as it is translated, looking for an (architecture-specific) call instruction (currently, we look for `call` on x86 and `bl` and `mov lr, pc` on ARM). If the block includes a call instruction, then we push the return address onto a shadow stack after each time that block executes.

Detecting the return from a function does not require any architecture-specific code. Before the execution of every basic block, we check whether the address we are about to execute is at the top of the stack; if so, we pop it. We only need to check the starting address of the basic block, because by definition a return terminates a basic block, so the return address will always fall at the beginning of a block.

We note that these techniques may fail if traditional call-return semantics are violated. For example, if a program emulated calls and returns by manually pushing the return address and using a direct jump, it would not be detected as a call. However, for non-malicious compiler-generated code, we have found that the algorithm described here works well.

5.3 Fixed String Searching

Searching for fixed strings is one of the most effective tools for finding useful tap points. Because we have to sift through many gigabytes of data that pass through tap points during any given execution, it is vital that string search be efficient in both time and space.

To satisfy these constraints, we developed `stringsearch`, a plugin which requires only one byte of memory per search string and per tap point. This one-byte counter tracks, for a given tap point, how many bytes of the search string have been matched by the data seen at the tap point so far. Whenever a byte is read from or written to memory, we can check what the next byte in the search string is using this position, and compare it to the byte passing through the tap point. If it matches, the counter is incremented; if it does not match, the counter is reset to zero. When the counter equals the length of the search string, we know that the search string has passed through the tap point, and we report a match. Note that because the counter is only one byte, our matcher only supports strings up to 256 bytes long; this cap could be easily raised to 65,536 bytes by using a two-byte counter, at the cost of doubling the memory requirements. Thus far, 256-byte strings have been more than sufficient.

This effectively implements a very simple deterministic finite automaton (DFA) matcher. Indeed, we believe that it should be possible to efficiently implement a streaming basic regular expression matcher that requires only an amount of memory logarithmic in the number of states needed to

represent the expression. We leave this generalization to future work, however.

5.4 Statistical Search and Clustering

Collecting bigram statistics on data that passes through each tap point is an efficient way to enable "fuzzy" search based on some training examples, as well as enabling clustering. To implement this we collect bigram statistics for all tap points seen in execution, as well as for the exemplar; the data seen at each tap point is thus represented as a sparse vector with 65,536 elements (one for each possible pair of bytes).

To search, we can then sort the tap points seen by taking the distance (according to some metric) from the exemplar. For our metric, we have chosen to use Jensen-Shannon divergence [18], which is a smoothed and symmetrized version of the classic Kullback-Leibler divergence [16] (also known as information gain). We also examined the Euclidean and cosine distance metrics, but found their performance to be consistently worse. Jensen-Shannon divergence between two probability distributions P and Q is defined as:

$$JSD(P,Q) = H\left(\frac{P+Q}{2}\right) - \frac{H(P)+H(Q)}{2}$$

where H is Shannon entropy.

Bigram collection is done by maintaining, for each tap point, two pieces of information: (1) the last byte that passed through the tap point, so that we can see bigrams that span a single memory access; (2) a histogram of all byte pairs seen at the tap point. The latter of these must be maintained sparsely: because our bigrams are based on bytes, a dense histogram would require 65,536 integers' worth of storage per tap point. Given that most of the executions examined in this paper contain upwards of 500,000 tap points, this would require more than 120GB of memory, which is clearly infeasible (and wasteful, since most of those entries would be zero).

Instead, we store the histogram sparsely, using a C++ Standard Template Library `std::map<uint16_t,int>`. This keeps memory usage down without sacrificing any accuracy, but it does introduce some extra complexity when processing the resulting histograms, as our search software must support sparse vectors rather than simple arrays. Because of this additional complexity, we opted to implement the search and clustering algorithms ourselves, after some initial prototyping using SciPy's `sklearn` toolkit.

Our clustering is based on the venerable k-means algorithm [31], but using the Jensen-Shannon divergence described in the previous section. As in the statistical search case, we use bigram statistics for our feature vectors. Initialization uses the KMeans++ algorithm [2], which helps guarantee that the initial cluster centers are widely separated. We evaluate the performance of this clustering compared to an expert labeling in Section 6.3.

Our statistical search tool is implemented in 246 lines of C++, and computes the Jensen-Shannon divergence between a training histogram (dense) and a set of sparse histograms. Our K-Means clustering tool is 481 lines of C++ code, and outputs a clustering of the sparse histograms using Jensen-Shannon divergence as a distance metric.[6] Both tools are multithreaded, which greatly speeds up the computation.

5.5 Finding SSL/TLS Keys

We have also implemented a PANDA plugin called `keyfind`, which locates tap points that write SSL/TLS master secrets. The SSL/TLS master secret is a 48-byte string from which an SSL/TLS-encrypted session's keys are derived; thus, if a tap point that writes the master key can be found, encrypted network traffic can be decrypted and analyzed.

The plugin operates on a recording in which a program initiates an encrypted connection to some server and an encrypted packet sent by the client (captured using, e.g., `tcpdump`). The `keyfind` uses each 48 bytes accessed at each tap point as a trial decryption key for a sample packet sent by the client. If the decrypted packet's Message Authentication Code (MAC) verifies that the packet was decrypted correctly then we can conclude that the tap point can be used to decrypt SSL/TLS connections made by the program under inspection. In Section 6.2.1 we show how this technique can be used to spy on connections made by the Sykipot malware, without performing a (potentially detectable) man in the middle attack.

6. EVALUATION

In this section, we evaluate the efficacy of our various tap point search strategies, described in Section 4, at finding tap points useful for introspection. Our experiments are motivated by real-world introspection applications, and so for each experiment we describe a typical application for the tap points found. Each experiment was also generally performed on a variety of different operating sytems, applications, and architectures in order to evaluate TZB's ability to handle a diverse range of introspection targets.

For the sake of readability, we have attempted to use symbolic names for addresses wherever possible in the following results. It is hoped that these will be more meaningful to the reader than the raw addresses, but we emphasize that debug information is in no way required for TZB to work.

6.1 Known Knowns

6.1.1 URL Access

Monitoring visited URLs is likely to be useful for host-based intrusion detection and prevention systems. For example, an IDS may wish to verify that outgoing requests were initiated by a human rather than malware on the users's machine, or match URLs visited against a blacklist of malicious sites. This poses a challenge for existing introspection solutions, as URL load notification is not generally exposed by a public API, and the data resides in a user application (the browser).

To find URL tap points, we created training executions by visiting a set of three URLs (Google, Facebook, and Bing) in the following operating systems and browsers: Epiphany on Debian squeeze (armel and amd64); Firefox 16.0.2, Opera 12.10, and Internet Explorer 8.0.7601.17514 on Windows 7 SP1 (x86); and WebPositive r580 on Haiku (x86). We used the `stringsearch` plugin to search for the ASCII and UTF-16 representations of the three URLs, and then validated

[6]The use of this distance metric is justified theoretically because Jensen-Shannon distance is a Bregman divergence [3] and empirically because our clustering typically converges after around 30 iterations.

Browser	Caller	PC
Deb Epiphany (arm)	`WebCore::KURL::KURL+0x30`	`WebCore::KURL::init+0x70`
Deb Epiphany (amd64)	`webkit_frame_load_uri+0xc3`	`WebCore::KURL::init+0x368`
Win7 IE8 (x86)	`ieframe!CAddressEditBox::_Execute+0xaa`	`ieframe!StringCchCopyW+0x50`
Win7 Firefox (x86)	`xul!nsAutoString::nsAutoString+0x1a`	`xul!nsAString_internal::Assign+0x1d`
Win7 Chrome (x86)	`msftedit!CTxtEdit::OnTxChar+0x105`	`msftedit!CTxtSelection::PutChar+0xb8`
Win7 Opera (x86)	`Opera.dll+0x2cf6c6`	`Opera.dll+0x142783`
Haiku WebPositive (x86)	`BWebPage::LoadURL+0x3a`	`BMessage::AddString+0x26`

Table 1: Tap points found that write the URL typed into the browser by the user.

each tap point found to ensure that it wrote only the desired data. The results can be seen in Table 1.

6.1.2 TLS/SSL Master Secrets

Monitoring SSL/TLS-encrypted traffic is a classic problem for intrusion detection systems. Currently, hypervisor- or network- based IDSes that wish to analyze encrypted traffic must perform a man-in-the-middle attack on the connection, presenting a false server certificate to the client. Not only does this require the client to cooperate by trusting certificates signed by the intrusion detection system, it also takes control of the certificate verification process out of the hands of the client—a dangerous step, given that many existing SSL/TLS interception proxies have a history of certificate trust vulnerabilities [15].

Instead of a man-in-the-middle attack, we can instead use TZB to find a tap point that reads or writes the SSL/TLS master secret for each encrypted connection, giving us a "man-on-the-inside". Because this secret must be generated for each SSL/TLS connection, if we can find such a tap point, it can then be provided to the IDS to decrypt and, if necessary, modify the content of the SSL stream.

To find the location of these tap points, we ran a modified copy of OpenSSL's `s_server` utility that prints out the SSL/TLS master key any time a connection is made. We then recorded executions in which we visited the server with each of our tested SSL clients, and noted the SSL/TLS master secret. Finally, we used `stringsearch` to search for a tap point that wrote the master key, and verified that the tap wrote exactly one master key per connection. For this test, we used: OpenSSL s_client 0.9.8 on Debian squeeze (armel), OpenSSL s_client 0.9.8 and Epiphany 2.30.6 on Debian squeeze (amd64), and Firefox 16.0.2, Google Chrome 23.0.1271.64, Opera 12.10, and Internet Explorer 8.0.7601 on Windows 7 SP1 (x86). The results are shown in Table 2.

There is one particular point of interest to observe in these results. In the case of Epiphany on Debian, we found that one level of callstack information was *not* sufficient—with only the immediate caller, the tap point contains more data than just the SSL/TLS master secret. This is because the version of Epiphany uses SSLv3 to make connections, and the pseudo-random function (PRF) used in SSLv3 has the form

$$MD5(SHA1(\ldots))$$

The other implementations instead use TLSv1.0, where the PRF has the form

$$MD5(\ldots) \oplus SHA1(\ldots)$$

This final XOR operation is done from a unique program point, so the tap point that results from it contains only TLS master keys. This points to a potential complication of using tap points for introspection: it is not always clear in advance how many levels of call stack information will be required.

We were successful in locating tap points for all SSL/TLS clients tested. We note that uncovering similar information using traditional techniques would have required significant expertise and reverse engineering of both open source and proprietary software.

6.1.3 File Access

Monitoring file accesses is a requirement for many host-based security applications, including on-access anti-virus scanners. Thus, locating a tap point at which system-wide file accesses can be observed is of considerable importance. However, because previous approaches to the introspection problem [11, 12] passively retrieve information from the guest and are not event-driven, they cannot be used in this scenario.

To find such a tap point, we created recordings in which we opened files in various operating systems. Specifically, in each OS we created 100 files, each named after ten successive digits of π. The operating systems chosen for this test were: Debian squeeze (amd64), Debian squeeze (armel), Windows 7 SP1 32-bit, FreeBSD 9.0, and Haiku R1 Alpha 3 (all on x86). We then searched for tap points that wrote strings matching the ASCII and UTF-16 encodings of the filenames using the `stringsearch` analysis plugin. The UTF-16 encodings were included because it was known that Windows 7 uses UTF-16 for strings pervasively, allowing us to surmise that on Windows URLs would likely be UTF-16 encoded. Finally, we looked at the tap points found by `stringsearch`, and validated them by hand.

The results are shown in Table 3. For most of the operating systems we had no difficulty finding a tap point that contained the name of each file as it was accessed. The one exception was Windows 7, where the most promising tap point not only wrote file results, but also a number of unrelated objects such as registry key names. As in the SSL case, the root cause of this was insufficient calling context: in Windows several different things fall under the umbrella of a "named object", and these were all being captured at this tap point. We found that four levels of calling context were sufficient to restrict the tap point to just file accesses; the "deepest" caller was `IopCreateFile` (which, despite its name, is used for both opening existing files and creating new ones).

6.2 Known Unknowns

6.2.1 SSL Malware

The need to snoop on SSL-encrypted connections arises in malware analysis as well. Two features distinguish this case from that of intercepting the traffic of benign SSL clients presented in the previous section. First, the ability to de-

Client	Caller	PC	Process
Deb OpenSSL (arm)	`tls1_generate_master_secret+0x9c`	`tls1_PRF+0x90`	openssl
Deb OpenSSL (amd64)	`ssl3_send_client_key_exchange+0x437`	`tls1_generate_master_secret+0x108`	openssl
Deb Epiphany (arm)	`md_write+0x74`	`md5_write+0x68`	epiphany
Deb Epiphany (amd64)	`md_write+0x60`	`md5_write+0x49`	epiphany
Haiku WebPositive (x86)	`tls1_generate_master_secret+0x65`	`tls1_PRF+0x14b`	WebPositive
Win7 Chrome (x86)	`chrome!NSC_DeriveKey+0x1241`	`chrome!TLS_PRF+0xa0`	chrome.exe
Win7 IE8 (x86)	`ncrypt!Tls1ComputeMasterKey@32+0x57`	`ncrypt!PRF@40`	lsass.exe
Win7 Firefox (x86)	`softokn3!NSC_DeriveKey+0xe85`	`freebl3!TLS_PRF+0xbb`	firefox.exe
Win7 Opera (x86)	`Opera.dll+0x2eb06e`	`Opera.dll+0x50251`	opera.exe

Table 2: Tap points found that write the SSL/TLS master secret for each SSL/TLS connection.

Target	Caller	PC
Debian (amd64)	`getname+0x13e`	`strncpy_from_user+0x52`
Debian (arm)	`getname+0x88`	`__strncpy_from_user+0x10`
Haiku (x86)	`EntryCache::Lookup+0x27`	`hash_hash_string+0x1b`
FreeBSD (x86)	`namei+0xd1`	`copyinstr+0x38`
Windows 7 (x86)	`ObpCaptureObjectName+0xcb`	`memcpy+0x33`

Table 3: Tap points found for file access on different operating systems.

crypt the traffic without a man in the middle is even more important: in contrast to benign clients, we cannot assume that malware will accept certificates signed by our certificate authority. Second, we cannot rely on having access to the server's master secret, as the server is under the attacker's control. This means that our previous strategy of using a simple string search for the master secret will not work here.

Instead, we located the tap point in the SSL-enabled malware using our `keyfind` plugin, which performs trial decryption on a packet sent by the malware using each possible 48-byte sequence written to memory as a key and verifies whether the Message Authentication Code is valid. Although this is much slower than a string match, it is the only available option, since the key is not known in advance.

To test the plugin, we obtained a copy of a version of the Sykipot trojan released around October 31st, 2012 [13] (MD5: `34a1010846c0502f490f17b66fb05a12`). We then created a recording in which we executed the malware; simultaneously, we captured network traffic using `tcpdump`. We noted that the malware made several encrypted connections to `https://www.hi-techsolutions.org/`, and provided one of the encrypted packets from these connections as input to the `keyfind` plugin. The plugin found the same tap point as the Windows 7 IE8 experiment described earlier, indicating that both the malware and IE8 likely use the same underlying system mechanism to make SSL connections. The key found was able to decrypt the connections contained in the packet dump.[7]

6.2.2 *Finding* `dmesg`

System logs are an invaluable resource, both for security and system administration. In an introspection-based security system, for example, one might want to find a tap point that contains the system's logs so that they can be stored securely outside the guest virtual machine. However, because the format of system logs is particular to each OS, we need some mechanism that can find tap points that write data that "looks like" a log based on an exemplar. The statistical search described in Section 4.2 is a good fit for this task: by training on the output of `dmesg` on one OS, we can find `dmesg`-like tap points on other systems.

To locate these system log tap points, we first created a training exemplar by running the `dmesg` command on a Debian sid (amd64) host and computing the bigram probabilities for the output. We then created recordings in which we booted five operating systems (Debian squeeze (armel), Debian squeeze (amd64), Minix R3-2.0 (i386), FreeBSD 9.0-RELEASE (i386), and Haiku R1 Alpha 3 (i386)), and computed the same bigram statistics. We then sorted the tap points seen in each operating system boot according to their Jensen-Shannon distance from the training distribution, and manually examined data written by the tap point for each of the top 30 results in each operating system. Table 4 shows, for each operating system tested, the tap point that we determined to be the system log, and its rank in the search results.

We can see that in all cases the correct result is in the top 10. There are two additional features of Table 4 that bear mentioning. First, the reader will note that the two Debian systems have a caller of "N/A". This is because the memory writes that make up `dmesg` are done in `do_syslog`, which is called from multiple functions. In these cases, including the caller splits up information that is semantically the same. We detected this case by noticing that several of the top-ranked results in the Linux experiments had the same program counter, and that they appeared to contain different sections of the same log. Second, the tap point found for Haiku was also incomplete—some lines were truncated. By using our tap point correlation plugin, we determined that we were missing a second tap point that was correlated with the main one; the two together formed the write portions of a `memcpy` of the log messages. Once this second tap point was included, we could see all the log messages produced by Haiku.

We also attempted to find an analogous log message tap point on Windows 7, but were not successful. This is a result of the way Windows logging works: rather than logging string-based messages, applications and system services create a manifest declaring possible log events, and then refer to them by a generated numeric code. Human-readable messages are not stored, and instead are generated when the user views the log. This means that there is no tap point

[7]The malware also has a second layer of encryption, which is custom and not based on SSL; we did not attempt to decrypt this second layer.

OS	Caller	PC	Kernel?	Rank
FreeBSD (x86)	`msglogstr+0x28`	`msgbuf_addstr+0x19a`	Yes	1
Haiku (x86)	`ring_buffer_peek+0x59`	`memcpy_generic+0x14`	Yes	1
Debian (arm)	N/A	`do_syslog+0x18c`	Yes	4
Debian (amd64)	N/A	`do_syslog+0x163`	Yes	4
Minix (x86)	`0x190005ee`	`0x190009d4`	No	8
Windows 7 (x86)	Not Found	Not Found	?	?

Table 4: Tap points that write the system log (`dmesg`) on several UNIX-like operating systems. All tap points were located in the kernel, except for Minix, which is a microkernel. We were unable to find a tap point analogous to `dmesg` in Windows.

that will contain log messages of the type used in our `dmesg` training, and the methods described in this paper are largely inapplicable unless a training example for the binary format can be found. However, because the event log query API is public [23], existing tools such as Virtuoso [11] might be a better fit for this use case.

Anecdotally, the ability to uncover a tap point that writes the kernel logs has also been useful for diagnosing problems when adding support for new platforms to QEMU. For an unrelated research task, we attempted to boot the Raspberry Pi [1] kernel inside QEMU, but found that it hung without displaying any output early on in the boot process. By locating the `dmesg` tap point, we discovered that the last log message printed was "Calibrating delay loop..."; based on this we determined that the guest was hung waiting for a timer interrupt that was not yet implemented in QEMU.

6.3 Unknown Unknowns: Clustering

To test the effectiveness of clustering tap points based on bigram statistics and Jensen-Shannon distance, we carried out an experiment that compared the clusters generated algorithmically to a set of labels generated by two of the co-authors manually examining the data. We created six recordings representing different workloads on two operating systems (Windows 7 and FreeBSD 9.0). From FreeBSD, we took recordings of boot, shutdown, running applications (`ps`, `cat`, `ls`, `top`, and `vi`), and a one-minute recording of the system sitting idle, for a total of four recordings. On Windows we created two recordings: running applications (`cmd.exe`, `dir`, the Task Manager, Notepad), and one minute of the system sitting idle.[8]

Next, we sampled a subset of the tap points found in each recording. Given that the vast majority of tap points do not write interesting information, we opted not to sample uniformly from the all tap points found. Instead, we performed an initial k-means clustering with $k = 100$, and then picked out tap points at various distances from each cluster center. We chose the tap point at σ standard deviations from the center, for $\sigma \in \{0, 0.5, 0.75, 1.0, 1.25, 1.5, 1.75, 2.0\}$ for a total of 2,926 samples[9]. Finally, we dumped the data from each of the sampled tap points, blinded them by assigning each a unique id, and then provided the data files to our two labelers. Each labeler independently assigned labels to each of the samples using the labels described in Table 5, and the two labelers then worked together to reconcile their labels.

Finally, we ran a k-means clustering with $k = 10$; 10 was chosen because it was a round number reasonably close to the number of labels our human evaluators gave to the data. We then used the Adjusted Rand Index [14] to score the quality of our clustering relative to our hand-labeled examples. The Adjusted Rand Index for a clustering ranges from -1 to 1; clusterings which are independent of the hand labeling will receive a score that is negative or close to zero. As can be seen in Table 6, our clustering did not match up very well on the hand-labeled samples. Note, however, that labeling criteria were selected without knowledge of the sizes of categories or whether or not the distance metric would effectively discriminate, so it is perhaps unsurprising that the correspondence is poor.

There is some hope, however. Regardless of the apparently poor clustering performance with respect to hand-labeling, we decided to determine if the clusters from our 100-mean clustering of FreeBSD's boot process contained new and interesting data and if finding that data would be facilitated by them. First, we determined to which of the clusters data from the FreeBSD's `dmesg` and filename tap points (found in Sections 6.2.2 and 6.1.3) was assigned. We were heartened to learn that these two text-like tap points had been sent to the same cluster. We proceeded to explore this cluster of approximately 5000 tap points, and found that, indeed, the vast majority of the tap points contained readable text of some sort. Further, in the course of about thirty minutes of spelunking around this cluster, we found not only kernel messages and filenames but a stone soup of shell scripts, process listings kernel configuration, GraphViz data, and so on. A selection of these tap point contents is provided in Appendix 9. We did not exhaustively examine this cluster, but plan to do so soon, as it appears to contain much of interest for active monitoring. If clustering has focused us on one out of 100 clusters, this is potentially a big savings.

6.4 Accuracy

Leaving aside the clustering results for the moment, the analyses implemented in TZB are extremely effective at helping to identify interposition points for active monitoring. In the evaluations based on string searching, we found that the number of tap points we had to look at manually was at most 262 (URLs under IE8) and in the best case we only had to examine two tap points (for SSL keys under Firefox, Opera, Haiku, and OpenSSL on ARM). The number of tap points that need to be examined is related to how widely the data is propagated in the system and how common the string be-

[8]Although we would have preferred to include Windows boot and shutdown recordings, at the time our replay system had a bug (now fixed) that prevented these recordings from being replayed.

[9]The alert reader will note that this is smaller than the 4,800 samples one would expect from taking 8 samples from 100 clusters in each of 6 recordings. This is because some clusters did not have very high variance, and so in many cases there were fewer than 8 samples at the required distance from the center.

Abbrv.	Description	Count
bp	binary pattern	2318
rd	repeated dword	400
mz	mostly zero	141
rq	repeated quadword	19
fnu	filenames unicode	8
woa	words ascii	8
wou	words unicode	7
inu	integers unicode	6
bu	binary uniform	5
ura	URLs ascii	5
rs	repeated short	4
fna	filenames ascii	2
rb	repeated byte	2
vr	very redundant	1

Table 5: Labels given to the sampled tap points by human evaluators, along with the number of times each occurred.

Recording	ARI
FreeBSD Apps	0.018
FreeBSD Boot	0.048
FreeBSD Idle	0.021
FreeBSD Shutdown	0.074
Win7 Apps	0.029
Win7 Idle	-0.003

Table 6: Quality of clustering as measured by the Adjusted Rand Index, which ranges from -1 to 1, with 1 being a clustering that perfectly matches the hand-labeled examples.

ing searched for is; thus, it is natural that URLs visited in the browser would appear in many tap points, whereas the SSL/TLS master key would not. Qualitatively speaking, we found that once the candidate tap points had been selected by `stringsearch` for a given execution, it took at most an hour to find one that sufficed for the task at hand.

For the `dmesg` evaluation, we also examined the quality of the results found for each operating system using the standard "Precision at 10" metric, which is just the number of results found in the top 10 that were actually relevant to the query. In this case, this is simply the fraction of results in the top 10 that appeared to contain the system log (even if it was incomplete). Based on this metric, the precision of our retrieval was between 20% (on Minix) and 100% (on Haiku). This means that if one looked at all of the top 10 entries, it is guaranteed that one would find the correct tap point.

7. LIMITATIONS AND FUTURE WORK

Although TZB is currently very useful for finding interception points for *active monitoring*, it is not currently usable in every scenario where introspection is needed. Because the interception points are triggered by executing code, they are only usable in *online* analysis. However, the need for introspection also arises in *post-mortem* analysis, specifically in forensic memory analysis. Whereas previous solutions such as Virtuoso [11] were able to operate equally well on memory images or live virtual machines, TZB is only applicable to the live case. In future work, we hope to combine Virtuoso-like techniques with TZB to produce offline programs that can locate in memory the buffers on which TZB's tap points operate.

Another limitation of TZB is its reliance on callstack information to locate interposition points. In current systems, keeping track of an arbitrary number of callers for each process is prohibitively expensive; although stack walking is faster (since it only needs to be invoked when the monitored code is executed), it is insecure, unreliable, and not available on every architecture. We hope to examine how existing solutions such as compiler modifications [32, 8] or dynamic binary translation [27] can be used to efficiently maintain the shadow stack information needed for TZB. We also note that Intel's Haswell architecture, which as of this writing has just been released, has hardware support for keeping track of calls and returns [33]; this would provide an excellent base on which to build a low-overhead security system based on TZB tap points.

Code generated at runtime (i.e., JIT or injected code) may make re-identification of a discovered tap point difficult or even impossible. Given the rise of languages that depend on JIT runtimes, better solutions are needed for this scenario, and the problem of how to make use of tap points in JIT code should be explored further.

Finally, as seen in Section 6.3, the clustering results are promising, but not yet fully developed. We hope to gain a better understanding of the data found in tap points and seek out better features and models for clustering in future work.

8. RELATED WORK

Although, to our knowledge, there is no existing work on mining the contents of memory accesses for introspection, we drew inspiration from a variety of sources. These can be roughly grouped into three categories: work on automating virtual machine introspection, research on automated reverse engineering, and efforts that examine memory access patterns, typically through visualization. In this section, we describe in more detail previous work in these three areas.

Virtual machine introspection has been targeted for automation by several recent research efforts because of the *semantic gap problem*: security applications running outside the guest virtual machine need to reconstruct high-level information from low-level data sources, but doing so requires knowledge of internal data structures and algorithms that is costly to acquire and maintain. To address this problem, researchers have sought ways of bridging this gap automatically. Virtuoso [11] uses dynamic traces of in-guest programs to extract out-of-guest tools that compute the same information. However, because it is based on dynamic analysis, incomplete training may cause the generated programs to malfunction. Two related approaches attempt to address this limitation: *process out-grafting* [30] moves monitored processes to the security VM while redirecting their system calls to the guest VM, allowing tools in the security VM to directly examine the process, while VMST [12] selectively redirects the memory accesses of tools like `ps` and `netstat` from the security VM so that their results are obtained from the guest VM. TZB extends these approaches by finding points in applications and the OS at which to perform active monitoring.

Based on the observation that memory accesses in dynamic execution can reveal the structure of data in memory, several papers have proposed methods for automatically deducing the structure of protocols [6, 19, 9], file formats [10, 20], and in-memory data structures [17, 28, 21]. One par-

ticular insight we have drawn from this body of work is the idea that the point in a program at which a piece of data is accessed, along with its calling context, can be used as a proxy for determining the type of the data. TZB leverages this insight to separate out memory accesses into streams of related data.

Finally, there has been some research on examining memory accesses made by a single program or a whole system, typically using visualization. Burzstein et al. [5] found that by visualizing the memory of online strategy games, they could identify the region of memory used to decide how much of the in-game map was visible to the player, which greatly reduces the work required to create a "map hack" and allow the player to see the entire map at once. Outside the academic world, the ICU64 visual debugger [22] allows users to visualize and modify the entire memory of a Commodore 64 system, enabling a variety of cheats and enhancements to C64 games. Although TZB does not use visualization, it shares with this previous work the understanding that memory accesses can be a rich source of information about a running program.

9. CONCLUSION

In this paper we have presented TZB, a system that automatically locates candidate memory accesses for active monitoring of applications or operating systems. This is a task that previously required extensive reverse engineering by domain experts. We have successfully used TZB to identify a broad range of tap points, including ones to dynamically extract SSL keys, URLs typed into browsers, and the names of files being opened. TZB is built atop the QEMU-based PANDA platform as a set of plug-ins and its operation is operating system and architecture agnostic, affording it impressive scope for application. This is a powerful technique that has already transformed how the authors perform RE tasks. By reframing a difficult RE task as a principled search through streaming data provided by dynamic analysis, TZB allows manual effort to be refocused on more critical and less automatable tasks like validation.

Acknowledgments

This material is based upon work supported in part by the National Science Foundation under grants no. CNS-1017265 and no. CNS-0831300, and the Office of Naval Research under grant no. N000140911042. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the Office of Naval Research.

10. REFERENCES

[1] Raspberry Pi: An ARM GNU/Linux box for $25. `http://www.raspberrypi.org/`.

[2] D. Arthur and S. Vassilvitskii. k-means++: the advantages of careful seeding. In *Proceedings of the ACM-SIAM symposium on Discrete algorithms*, 2007.

[3] A. Banerjee, S. Merugu, I. S. Dhillon, and J. Ghosh. Clustering with Bregman divergences. *J. Mach. Learn. Res.*, 6, Dec. 2005.

[4] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*, 2005.

[5] E. Bursztein, M. Hamburg, J. Lagarenne, and D. Boneh. OpenConflict: Preventing real time map hacks in online games. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.

[6] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the ACM conference on Computer and communications security*, 2007.

[7] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM SIGARCH Computer Architecture News*, 39(1), 2011.

[8] T. Chiueh and F. Hsu. RAD: a compile-time solution to buffer overflow attacks. In *International Conference on Distributed Computing Systems*, 2001.

[9] W. Cui, J. Kannan, and H. J. Wang. Discoverer: automatic protocol reverse engineering from network traces. In *Proceedings of the USENIX Security Symposium*, 2007.

[10] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz. Tupni: automatic reverse engineering of input formats. In *Proceedings of the 15th ACM conference on Computer and communications security*, 2008.

[11] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2011.

[12] Y. Fu and Z. Lin. Space traveling across VM: Automatically bridging the semantic-gap in virtual machine introspection via online kernel data redirection. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2012.

[13] K. Gilbert. Hurricane sandy serves as lure to deliver Sykipot. `http://securityblog.verizonbusiness.com/2012/10/31/hurricane-sandy-serves-as-lure-to-deliver-sykipot/`.

[14] L. Hubert and P. Arabie. Comparing partitions. *Journal of Classification*, 2(1), 1985.

[15] J. Jarmoc. SSL/TLS interception proxies and transitive trust. In *Black Hat Europe*, March 2012.

[16] S. Kullback and R. A. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22, 1951.

[17] J. Lee, T. Avgerinos, and D. Brumley. TIE: Principled reverse engineering of types in binary programs. In *Network and Distributed System Security Symposium*, 2011.

[18] J. Lin. Divergence measures based on the Shannon entropy. *IEEE Trans. Inf. Theor.*, 37(1), Sept. 2006.

[19] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Network and Distributed Systems Symposium*, 2008.

[20] Z. Lin and X. Zhang. Deriving input syntactic structure from execution. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of software engineering*, 2008.

[21] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution.

In *Network and Distributed System Security Symposium*, 2010.

[22] mathfigure. ICU64: Real-time hacking of a C64 emulator.

[23] Microsoft Corporation. EvtQuery function. `http://msdn.microsoft.com/en-us/library/windows/desktop/aa385466(v=vs.85).aspx`.

[24] T. Müller, F. C. Freiling, and A. Dewald. TRESOR runs encryption securely outside RAM. In *Proceedings of the 20th USENIX conference on Security*, 2011.

[25] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *IEEE Symposium on Security and Privacy*, 2008.

[26] D. Rumsfeld. DoD news briefing - Secretary Rumsfeld and Gen. Myers. February 2002.

[27] S. Sinnadurai, Q. Zhao, and W. Wong. Transparent runtime shadow stack: Protection against malicious return address modifications. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.120.5702`, 2008.

[28] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Network and Distributed Systems Symposium*, 2011.

[29] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Information systems security*. 2008.

[30] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu. Process out-grafting: an efficient "out-of-vm" approach for fine-grained process execution monitoring. In *Proceedings of the ACM conference on Computer and communications security*, 2011.

[31] H. Steinhaus. Sur la division des corp materiels en parties. *Bull. Acad. Polon. Sci*, 1, 1956.

[32] Vendicator. Stack shield: A "stack smashing" technique protection tool for Linux. `http://www.angelfire.com/sk/stackshield/`.

[33] Z. Yan. perf, x86: Haswell LBR call stack support. `http://lwn.net/Articles/535152/`.

APPENDIX

A. SAMPLE TAP POINT CONTENTS

Here, we reproduce a selection of tap points from the same cluster as dmesg and filename tap points.

```
/etc/rc.d/ipfw/etc/rc.d/NETWORKING/etc/rc.d/netwait/etc/rc
.d/mountcritremote/etc/rc.d/devfs/etc/rc.d/ipmon/etc/rc.d/
mdconfig2/etc/rc.d/newsyslog

r=/sNsnWs/fuiebu/ r=ceremdsecd_t_co_artpachg=tSooadSebaabf
/faa_N_=_peOfA=fA=feTr=tul.n=_eo/.b_Yt_vtectvifat=a=-sd_Ee
Ofu=u_0y:nF:tRseeeeEfciOtmdtuinlrlrrlpp/nppfpcepinl=l=.llN
lNlglllpl_.4l_l_2/l_l_22lileldlylo- 21laltlat=rrrsbgrskgni/

russian|Russian Users Accounts:  :charset=KOI8-R:        :
lang=ru_RU.KOI8-R:     :       :passwd_format=md5:     :co
pyright=/etc/COPYRIGHT:      :welcome=/etc/motd:     :sete
nv=MAIL=/var/mail/$,BLOCKSIZE=K,FTP_PASSIVE_MODE=YES:

nss_compat.so.1dhclientShared object ‘‘nss_compat.so.1’’ n
```

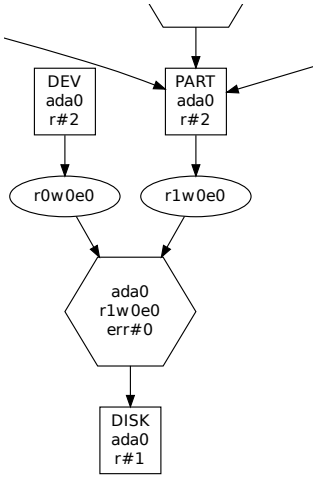

Figure 4: Detail from rendering of Graphviz file captured from a FreeBSD boot tap point, apparently depicting disk geometry

```
ot found, required by ‘‘dhclient’’nss_nis.so.1dhclientShar
ed object ‘‘nss_nis.so.1’’ not found, required by ‘‘dhclie
nt’’nss_files.so.1dhclientShared object ‘‘nss_files.so.1’’

digraph geom {
z0xc1d8de00 [shape=box,label=’’PART\nada0\nr#2’’];
z0xc1f4f640 [label=’’r1w0e0’’];
z0xc1f4f640 -> z0xc1e9eb00;

/sbin/in/bin/sh/bin/stt/sbin/sysctl/bin/ps/sbin/sysctl/sbi
n/rcorde/bin/cat/sbin/md/sbin/sysctl/sbin/sysctl/bin/ken/s
bin/dumpon/bin/ln/bin/ps/sbin/sysctl/sbin/sysctl/sbin/sysc
tl/sbin/sysctl/bin/ps/bin/dd/sbin/sysctl/bin/dat/bin/df/sb

/boot/kernel/kernel00000000-0000-0000-0000-000000000000000
00000-0000-0000-0000-0000000000000000000000-0000-0000-0000-0
00000000000993c915d-3e9f-11e2-a557-525400123456993c915d-3e
9f-11e2-a557-525400123456/boot/kernel/kernel/boot/kernel/k
...
modulesoptions       CONFIG_AUTOGENERATED
ident   GENERIC
machine i386
cpu     I686_CPU
cpu     I586_CPU
cpu     I486_CPU

<mesh>
  <class id=’’0xc10362c0’’>
    <name>FD</name>
  </class>
  <class id=’’0xc1009a80’’>

#!/bin/sh
#
# $FreeBSD: release/9.0.0/etc/rc.d/newsyslog 197947 2009-1
0-10 22:17:03Z dougb $
...
set_rcvar()
{
        case $# in
        0)
                echo ${name}_enable
                ;;
        1)
```