Automatically Bridging the Semantic Gap using C Interpreter

Hajime Inoue, Frank Adelstein, Matthew Donovan, Stephen Brueckner

***Abstract*—We describe min-c, a C interpreter that solves the generalized problem of the "semantic gap". The semantic gap exists in virtual machine introspection (VMI) and in volatile memory forensics because there is not a native hardware environment. For example, a pointer in a data structure in a process cannot be used without translation to a physical address, a function of the native hardware and operating system. The usual solution is to build an OS interface library to provide the necessary translations. This is brittle as it must constantly track OS versions. Min-c solves this problem by enabling automatic generation of the OS interface library using native OS code itself, or debugging symbols when source is not available. We describe the design of min-c and our method for automatically building the semantic interface database required for type interpretation for both Linux and Windows OSs.**

***Index Terms*—Forensic Memory Analysis, Virtual Machine Introspection, Semantic Gap, Volatile Memory, C Interpreter**

I. Introduction

IN volatile memory forensics and virtual machine introspection (VMI) it is necessary to interpret, at a high level, the state of a system which has only been recorded at a very low level. Effective techniques for doing this are increasingly necessary. Most forensics investigations currently are centered on non-volatile storage (hard disks). Hard disk capacities are enormous, however, and the use of encrypted file systems is increasing. Volatile memory can often provide evidence, such as encryption keys, which makes analysis of non-volatile storage faster and easier.

The motivation for VMI is quite different. Sophisticated, stealthy malware (e.g., rootkits) can subvert operating systems entirely, disabling and hiding from the most sophisticated computer security software. It is always possible for the malware to discover, disable, or hide from security software because it runs on the same machine. In light of this, many have proposed and demonstrated approaches based on VMI. This "out-of-the-box" technique allows security software to run in a trusted environment completely outside the OS and the applications it observes.

Both techniques require one machine to interpret the low level state of another machine. VMI applications (we refer to the VM hosting the introspecting application as the host) use the hypervisor to directly access the state of another VM's (the guest) virtual hardware, including the processor, memory, and devices (e.g., disk and network). In volatile memory forensics, the available resource is a core-dump or raw memory image.[1] The difficulty in interpreting this low-level data into a high-level model of the guest system's state is referred to as the semantic gap [1].

The two research communities focusing on this problem have, until recently, been quite distinct. However, we noticed in the course of our VMI research that the problem posed by VMI is identical to that posed in forensics. The goal of our research was to develop a model of the guest's kernel memory space using the semantics of its operating system – the common problem in forensics and VMI. VM introspection libraries supply processor state, and accessing the file system is fairly straightforward [2]. Network operations can also be captured and interpreted easily from outside the VM. The VMI problem is more difficult in that it must not cause side-effects within the guest, and it must also run efficiently, so that real-time monitoring is possible. We therefore concentrate on VMI in this paper, although our technique applies generally to volatile memory forensics as well.

Others (described in Section V) have addressed the semantic gap problem but each proposed solution has limitations. Operating systems may be categorized into major classes (e.g., Windows XP, Vista, Linux 2.4, Linux 2.6), but there is significant variation within each major class that previous efforts do not address. First, they do not account for versioning. Modern operating systems are patched quite frequently, requiring either modification or at the very least, recompilation, of the introspection software. Second, in open source operating systems such as Linux, kernels are customized by distributions or even by individuals themselves.

Due to these two factors, the semantic gap-bridging software

H. Inoue, F. Adelstein, M. Donovan, and S. Brueckner are with ATC-NY, Ithaca, NY 14850 USA (e-mail: {hinoue, fadelstein, matthew, steve} @atc-nycorp.com). This research was supported by the Air Force Research Lab under award FA8750-07-C-0106. The opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the Air Force Research Lab.

[1] For the purposes of this paper, we will refer to the analysis machine as the host and the machine where the memory image was gathered as the guest.

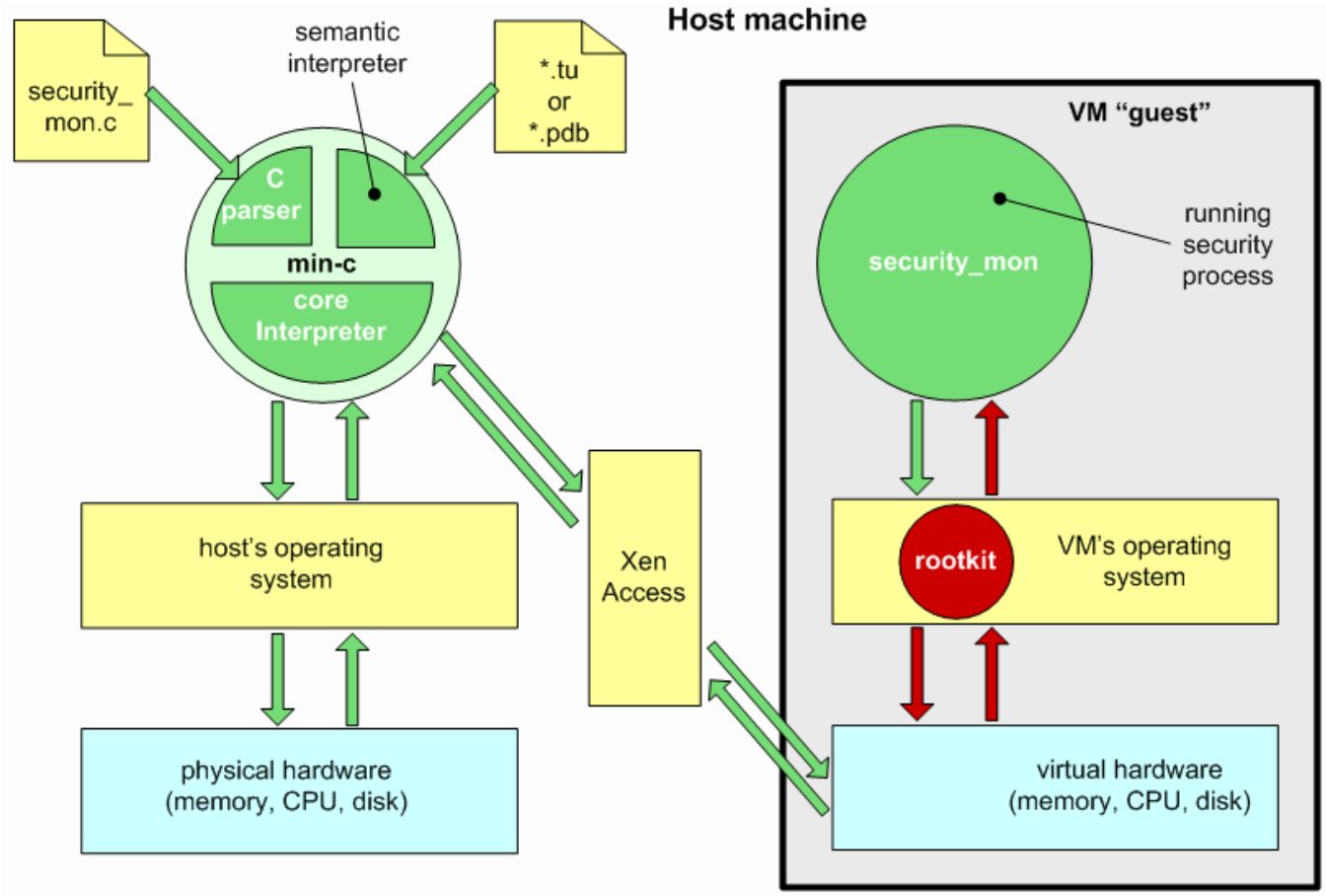

Fig. 1. A comparison of a security monitor using VM introspection with min-c (left) and similar software hosted in the traditional manner (right).

of previous efforts have been prototype implementations written for specific kernel versions that are brittle in the face of changes to those kernels. In addition, each previous effort required manual labor to acquire detailed knowledge of each guest's data types (including the fields of aggregate types) and magic numbers (i.e., memory locations, including relative offsets within aggregate types). This manual labor can include inspection of available files (e.g., symbol tables, header files, source code), reverse engineering, kernel debugging, and trial-and-error coding. For example, Volatility [8], the most popular tool for volatile memory forensics, only supports Windows XP; it does not yet support Windows Vista or Windows 7.

Based on the shortcomings of previous efforts, we determined that a useful solution for bridging the semantic gap should be both general and automatic. That is, it should enable us to run any version or distribution of a major OS class without recompilation of the introspection code and without the need for manual intervention in the process.

Finally, a solution that bridges the semantic gap should minimize the distinction between guest and host. It should be easy to reuse and port security software. Developers should not need to learn an introspection API or language, as previous solutions have required. In short, we want our introspection layer to be invisible—host code should look and run as if it were in the guest.

Given these desiderata, we implemented a C interpreter, linked with an introspection library. It currently runs a large subset of the C90 standard.[2]

In addition, we have generated semantic reconstruction libraries that allow us to automatically locate and properly interpret data structures in both the Linux and Windows kernels. We call our semantic gap-bridging software min-c, which integrates our C interpreter and semantic reconstruction libraries. Fig. 1 shows how we use introspection and min-c to achieve our objectives and compares them to traditional non-introspective security software running inside a guest.

Min-c is an abbreviation for "EXAMIN-C." EXAMIN is a commercial project with the goal of developing a testing platform for containing, triggering, analyzing, and reverse engineering stealthy malware. It is a VM-based workbench based on the Xen hypervisor that employs VM introspection to provide high-assurance detection of stealthy malware on both Windows and Linux platforms. EXAMIN incorporates a number of practical tools based on our introspection

[2] C90 was chosen for its ease of implementation. We may switch to C99 in the future. The interpreter does not currently support some data types, such as ones we have not encountered in the Linux and Windows kernels (e.g., floating point).

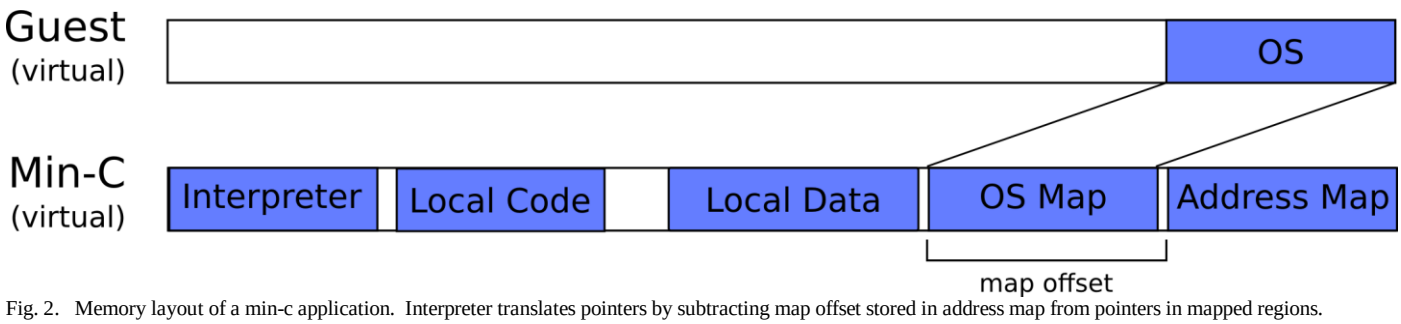

Fig. 2. Memory layout of a min-c application. Interpreter translates pointers by subtracting map offset stored in address map from pointers in mapped regions.

techniques, including integrity monitors and cross-view checkers.

Min-C forms the core of EXAMIN. In the rest of the paper, we describe how min-c solves the various problems involved in practical VM introspection and bridging the semantic gap. We begin by presenting our min-c design. We first describe how we automate the gathering of kernel-specific semantic information and then how we integrate this information with our C interpreter. Next, we describe our current applications of min-c. Finally we discuss the limitations of our approach and describe future and related work before concluding the paper.

II. Design

The min-c interpreter consists of the following four components, each of which we subsequently describe in detail:

1. the introspection library,
2. the C parser,
3. the core interpreter, and
4. the semantic interpreters.

We used the open source XenAccess library to provide VM introspection of the Xen hypervisor [3]. We modified XenAccess to provide the ability to memory map a range of contiguous virtual memory from the guest into the host's address space. This is the principal introspection mechanism. XenAccess also provides access to files, enabling volatile memory forensics investigations as well.

We built the C parser with the aid of the TreeTop parsing library for the Ruby programming language [4]. The C parser reads in the source text and generates custom bytecode. Our parser performs type checking, but our custom bytecode instruction set does not embed type information such as size or offset information. Instead, type information is retrieved (for host datatypes) or calculated (for guest datatypes) during execution, because the version of the OS is not determined until runtime.

The core interpreter executes the bytecode generated by the parser. It uses the XenAccess library to map guest kernel memory into the host's address space and then operates on that memory similar to the way it operates on unmapped memory. The only exception is in the use of pointers: the address space mapped into the host makes pointers inconsistent. A variable referred to by a pointer in the guest has a different address on the host. The min-c interpreter keeps track of which portions of memory are mapped in from the guest and translates pointer addresses appropriately.

Fig. 2 shows how the map of the physical memory of guest OS is mapped into the process space of min-c. In the figure, the OS memory is contiguous in physical memory, which is usually the case. It is also possible to map the virtual address space of a user process, which is usually not physically contiguous, into a min-c application. A min-c application has five

logical memory areas: the mapped memory from the guest, the min-c interpreter itself, the min-c application's code (local code), its heap (local data), and the address map.

The address map is used by the interpreter to translate pointer addresses. It is a table which describes the mapping between the virtual address space of the guest and the virtual address space of the min-c application. When a min-c application reads a pointer, the interpreter consults the table to determine if it is in a guest-mapped space. If it is, then the pointer is read from the mapped space and then the map offset between the two spaces is added so that the new pointer is accurate. Note that the interpreter must understand that the value being read is a pointer. Applications that treat pointers as integers will not have these values properly translated. Thus, developers need to take particular care with types or memory corruption will immediately result.

Our semantic interpreters provide type and layout information for data structures within the guest. Semantic view reconstruction begins with an address and a type. If we have information about the type, we can then understand the region of memory specified by the address in terms of operating system semantics. If the type contains a pointer, we can then recursively apply this procedure to the address and type referred to by the pointer. In this way, we can rebuild the semantic meaning of the address space within the guest.

In Section V, we describe the manual and debugger-oriented procedures previous efforts have used to generate a semantic view of the guest's memory. As discussed in Section I, the procedures used in previous efforts are brittle and difficult to use with heterogeneous systems. In order for our semantic interpreters to automatically reconstruct a semantic view of the observed guest kernel, we need the data structure layout created by the compiler. For Linux, compiling the

observed kernel with debug symbols would give us access to this information, but unfortunately this would change the resulting image. For Windows, source code is not available so recompilation is also not an option. We extract the data structure information using a unique procedure for each major class of operating system, as follows.

For Linux, the GCC compiler can output a kernel's intermediate representation as a flat file database (.tu) in text format using the --fdump-translation-unit flag [5]. Since source code and kernel configurations are available for each Linux distribution, we can recompile each kernel of interest in this manner a single time, and use the resulting .tu files for all subsequent introspection activities. We do not need to replace the original guest kernel with the recompiled one because they are identical (We delete the recompiled one, saving only the .tu files generated by the compilation process). The objective is to generate the equivalent of debugging information without having to compile the kernel with a debugging flag. The .tu files give us access to the type, size, and layout information needed to automate semantic view reconstruction. Addresses are not available in the intermediate representation (these are generated by the linker), so we use the System.map file generated by the kernel compilation process (and usually exported in the /boot directory) to obtain them. The System.map file gives us exported addresses, but does not give us other important addresses, such as the location of the system call table, which we obtain through forensic processes that are automatable for each major kernel version.

For Windows, the compiler generates a Program DataBase file (.pdb) that contains the necessary symbol and type information [6]. An executable file (.exe or .dll) stores the name of its associated .pdb file as well as the version (specified by a globally unique identifier and age value). PDB files can be made available by an application developer, or in the case of Microsoft's executables (such as the Windows XP, Vista, and 7 kernels), are downloadable from Microsoft's Windows Symbol Server. Dynamic Link Libraries (DLLs) export public symbols for use by other applications but non-public symbols are stripped out during the compilation process. PDB files contain both public and private symbol information, such as addresses, as well as function signatures and their locations. Addresses are stored as Relative Virtual Addresses (RVAs) that are simply offsets from the start of the file when loaded into memory [7]. PDB files also contain data type information similar to that available in Linux .tu files.

Not all relevant information is provided by PDB files. Many internal data structures are not described, nor are many addresses which are useful for VMI or forensics. When structures or addresses are not available, developers in min-c can supplement this with files written in C. This is a key advantage of min-c, since it allows developers to augment the interface library and analysis scripts in the same language they use to write native OS code itself, and in a way that makes calls to the interface library implicit, as we show in the next Section.

Fig. 1 shows how four parts (the introspection library, C-parser, core interpreter, and semantic interpreter) compose the min-c interpreter and conduct semantics-aware VM introspection of a guest's kernel. A complete program is created when min-c extracts, interprets, and combines the source code (.c) file with the semantic information database (.tu or .pdb) files. The core interpreter uses local resources to execute, but redirects introspection queries to the guest via XenAccess. It accomplishes this by tracking and maintaining consistency of separate data types and pointers for each domain.

TABLE I
PROGRAMS FOR LISTING LOADED MODULES
WITHIN A GUEST VM RUNNING LINUX 2.6 KERNEL

XenAccess Specific Partial Listing

```
xa_read_long_sym(&xai, "module", &next_module;
list_head = next_module;

while(1) {
   memory=xa_access_virtual_address(&xai, next_module, &offset);
   if(memory == NULL) {
      perror("failed to map memory for module list pointer");
      goto error_exit;
   }
   memcpy(&next_module, memory+offset, 4);

   if(list_head == next_module)
      break;

   /* Note – the module struct that we are looking at has a string directly
following the next/prev pointers. This is why you can just add 8 to get the
name. See include/linux/module.h for more details. */
   name = (char *)(memory+offset+8);
   printf("%s\n", name);

   munmap(memory, xai, page_size);
}
```

Min-c Equivalent

```
/* Pull address from System.map */

extern struct list_head module* modules;
struct list_head* next_module = modules;

while(1) {
   struct module tmp;
   next_module = next_module->next;
   if(modules == next_module)
      break;

   tmp = list_entry(next_module, struct module, list);
   puts(tmp->name);
   puts("\n");
}
```

On the top is a partial source listing that uses only the XenAccess library. On the bottom is the mini-c version. Note: We ignore locking in this example.

III. CURRENT APPLICATIONS

We currently use min-c within our EXAMIN platform for

cross-view checking and integrity checking. Rootkits hide their presence from both user-space and kernel-space applications. Cross-view checking tools compare the state of the guest as reported from within (the guest) with that reported from without (the host) using VM introspection. Differences often indicate that stealthy malware has infected the host.

The min-c interpreter makes it much easier to write cross-view checking tools. To illustrate, consider Table I, which shows the listing for VM introspection code that outputs the list of modules running in a Linux system. Using only the XenAccess VM introspection library, the code is awkward, and relies on hardcoded offsets for linked-list pointers and the name string. Min-c, however, automatically reconstructs a guest kernel's datatypes and offsets, so its code is identical to that written for a kernel module. It is shorter, easier to understand, easier to write, and will run on many different versions of Linux, since the structure is interpreted at runtime, instead of compiled.

Our integrity checking tools monitor portions of the guest kernel's memory to detect tampering. During execution, the kernel code and many of its variables should not change frequently after initialization. For our EXAMIN implementation, integrity checking tools repeatedly poll code and selected structures, such as the system call table or interrupt descriptor table, for modification. Changes can indicate a rootkit infection.

Our integrity checking tools can also take advantage of min-c's semantic reconstruction capabilities. Rather than simply detecting a change to a data structure, we can describe the change. For example, when the system call table is "hooked," min-c can provide the name of the specific call that was altered and locate the memory space to which it now points.

IV. Discussion

The min-c interpreter effectively achieves our goals; it automatically bridges the semantic gap in a manner that is general to major kernel classes. Users do not have to learn a new API or language, and can use the native language (C) of the operating system to write scripts that look like kernel code running in the guest.

We have rewritten the larger examples provided with XenAccess in min-c style.[3] We have scripts identical to code written for the kernel, which execute properly for multiple versions of Windows and Linux. Our scripts list current processes and drivers (modules in Linux). We have developed cross-checkers for the system call tables for Windows and Linux which will inform administrators when the system call table has been modified, and which system calls have been hooked, which can help identify which rootkit is responsible.

This functionality forms the basis of our EXAMIN system, which is intended as a malware incubator. We can easily write new monitoring scripts for the system. Debugging is quite easy; we find that we can write Windows drivers and Linux modules to monitor kernel state directly, and then run them in min-c.

EXAMIN is clearly beneficial as a security and reverse engineering tool. It also can be useful as a tool to aid digital forensic analysis, in particular analysis involving live systems and volatile data, such as live memory. The interest in live memory analysis is growing rapidly, and while many advances have been made in recent years, there are relatively few tools to bridge the semantic gap.

Volatility [8] is probably the best known tool to conduct memory analysis, but until recently it relied on a pre-existing memory image. Simply getting the memory image can be challenging [9]. We became aware during the implementation of min-c that the Georgia Tech group responsible for XenAccess added hooks that enable Volatility to access live VMs, giving it similar introspective capabilities to min-c [10]. Similarly, XenAccess also supports access to memory dump files. While Volatility currently has more analysis tools than min-c, we believe that the min-c approach is superior because new tools for Volatility must be written in Python and use explicit translation libraries. Min-c can make writing new tools much easier. New system analysis tools will use or reuse code that is almost identical to the equivalent kernel C code. It also allows a single tool to target multiple versions of an OS, because the interpreter links in the appropriate translation library at runtime. With Volatility, this is not possible.

Because EXAMIN uses VM introspection, it has essentially no impact on the running system and is very unobtrusive. These qualities are highly desirable for forensic analysis [9]. EXAMIN can help analysts conduct an investigation of a running system to find data that may only reside in memory, or may locate data that are essential for a traditional disk analysis, such as whole-disk encryption keys, whose absence render the data on a disk useless.

A. Limitations

There are still several limitations to min-c that hinder its application to other problems.

First, we cannot read guest memory that is paged out to disk. This is not a problem with our current EXAMIN objective of kernel monitoring because kernel memory for the kernel structures that are required by our tools is never paged out.[4] To monitor guest memory for user-space applications would require modification of the guest: injecting code to induce page faults, causing the desired application pages to be read back into core. Although this is possible, our current

[3] The minor examples supplied by Xen Access are implicitly provided by min-c functionality.

[4] Volatility does not have this ability, either.

goals are to perform introspection without directly tampering with the guest, allowing security services to remain invisible.

Currently, no volatile forensics applications support examination of paged-out memory, either. It may be more straightforward to implement support for this for forensics, where everything is on a file system, than on an executing system where memory and the file system monitoring must maintain consistency.

Also, because our introspection method is based on polling, we cannot detect changes when they are quickly reversed. This possibility becomes less likely if our scripts poll more often, but the cost is decreased performance. We can fix this problem using memory access alerts (Section IV-B), but this will require modifications to the Xen hypervisor.

Because persistent malware is much more likely to be detected through file system virus checkers or similar software, we believe memory-only rootkits are becoming more common, and the min-c approach is most effective against them.

Third, the performance of min-c is limited by the fact it is interpreted rather than compiled. There are two reasons for choosing to run interpreted. First, data structure layout differs depending on patch-level in Windows and kernel configuration and compiler version in Linux. It is convenient to have one introspection application script that is useful for every guest of a major class instead of requiring new compiled versions for every kernel upgrade. This could be partially mitigated by having a just-in-time (JIT) compilation system, but this still would not mitigate the second problem: that of pointer translation. We map guest kernel memory into the min-c address space on the host using Linux's mmap. Our interpreter properly converts pointer values from addresses on the guest to addresses in the min-c address space. Because the introspection library does not allow us to specify a target address for mmap, we cannot calculate this address ahead of time. We could fix this with more complicated logic in a JIT compiled approach if greater performance is required.

Fourth, min-c is strictly a C interpreter. It does not interpret C++ code, nor can it import C++ datatypes (classes) for either Linux or Windows. This is acceptable for our current applications because the target is the kernel, which is typically written in C.

B. Future Work

Our min-c interpreter is still incomplete. There are still several features we believe will make it more useful.

Two features would enhance min-c for both digital forensics use and VMI:

JIT Compilation

We plan to port min-c to the LLVM framework [11]. LLVM is a toolkit for writing interpreters, virtual machines, and compilers. LLVM would vastly increase performance and allow us to support many languages.

OS Fingerprinting

At the moment, the operator must specify the correct version and patch level for the guest in order for the interpreter to identify the appropriate .pdb or .tu files needed to interface with it. We intend to automate this so that version information is automatically deduced by min-c on startup. On Linux, this information is usually available in the /proc/version file and is represented in memory by the init_uts_ns variable. Unfortunately, this variable's address differs by version and configuration, so it is not straightforward locate it and perform the check. The process is easier on Windows because the executable files themselves store the name and version of the associated .pdb file. It is a simple manner to parse the executable on disk to correctly identify the correct .pdb file. Pagel has described an effective method for fingerprinting Windows [12]. This is particularly important for digital forensics. While it is likely that operators will know what operating systems are running in their VMs, forensics investigators often receive images with no other information describing them.

Three other potential features would be VMI specific, and would require modifications to the hypervisor or XenAccess layer:

Synchronized Access

Access to many OS data structures is synchronized using locks. If a structure is locked (being modified) when we attempt to read it, it may be in an inconsistent state, causing our interpreter to make incorrect semantic interpretations. An appealing approach is to lock the data structures from the interpreter, allowing us to properly access these extended data structures. Our interim method is to pause the guest and check that the data structure is unlocked. If so, we read it, otherwise we briefly execute the VM and try again. Note that this procedure can act as a spin lock.

Memory Access Alerts

A highly desirable feature would be the ability to raise an alert when the guest writes to, reads from, or executes within a specified address range. This alert could cause the execution of an arbitrary script. It would also obviate the need for polling and enable EXAMIN's integrity tools to immediately discover when changes are occurring and prevent rather than detect intrusions. Memory alerts on read or execute operations could act as breakpoints, allowing min-c to operate as a scriptable debugger. VMware's VMSafe introspection library supports memory triggers, but requires that a VM be booted in a special introspection mode [13]. It is impossible to start or stop monitoring during VM execution. We prefer XenAccess's ability to start and stop monitoring of any VM at any time.

Replay

Finally, we consider the ability to rollback execution of the guest to any point in the past to allow examiners, when anomalies are discovered, to hunt down their origins. This would be similar to the ReVirt system [14], but with usability perhaps more similar to UndoDB [15].

V. Related Work

Most research on the semantic gap is specific to VMI. The Volatility project appears to be the basis of digital forensics research on volatile memory research. There has been recent work towards automated generation of this OS library. Case, Marziale, and Richard [16] demonstrated automatic generation of the Volatility OS library using .pdb files. Okolica and Peterson [17] use a similar strategy. Our work is more general, in that we also enable specification using C, which allows us to support Linux and other operating systems. The C preprocessor in particular allows us to support multiple OSs in a compact, clear way.

The VMI techniques described in this paper require access to a guest VM's state via the hypervisor. The open source project XenAccess [3] facilitates this process for the Xen hypervisor, and VMware's VMsafe [18] provides access to information from some of VMware's hypervisors. Both projects are fairly young and do little more than acquire the state of virtual hardware and do little to bridge the semantic gap with the guest OS, although XenAccess includes some sample modules that interpret structures in kernel memory.

The problem of creating high-level semantics from low-level hardware information acquired by VM introspection was identified by Chen and Noble [1], who applied the term "semantic gap." An early solution for bridging the semantic gap was implemented by Garfinkel and Rosenblum [19] in their Livewire prototype. The approach used a Linux crash dump analysis tool [20] as an "OS interface library." However, this approach applies only to kernels which have been compiled using non-standard flags (including debug symbols). Our approach allows us to run the same kernels that ship with standard Linux distributions, without requiring recompilation. Also, their tools are specific to Linux while our approach works with Windows kernels as well.

Many techniques useful in VM introspection are common to digital forensics. Several toolkits have been written that interpret physical memory images. Toolkits include idetect [21], Windows Memory Forensic Toolkit [22], the Volatility Framework [8], VADTree [23] and FATKit [24]. Others try to search memory directly, without fully rebuilding semantic representations. This is called "memory carving"; Schuster's DFRWS '06 paper is an example [25]. Carving is typically focused on a limited set of non-kernel datatypes and does not deal with memory layout, and is therefore insensitive to specific OS version and patch level. However, it is much less capable than the previous approaches that try to bridge the semantic gap by interpreting kernel structures, and therefore only support a limited range of OSs.

A common technique to bridge the semantic gap is to locate known structures in memory (by symbol table lookup, access to source code, or by scanning memory for matches) and then traverse and interpret these structures. This technique is used by [26], [27], [28], [29], [30], and [31]. These efforts rely on the manual process of locating ``magic numbers'' (structures addresses, their internal data types, and relative offsets), and writing the equivalent of kernel code to traverse and interpret them. Furthermore, this manual process must be repeated for different kernels and kernel versions, which frequently change due to new releases, patches, re-compilation, etc. This challenge is acknowledged by Hoglund [32] and Jiang et al. [2].

EXAMIN's feature set is similar to a research system built by Jiang, Wang, and Xu [2]. Their paper also details the difficulties in bridging the semantic gap, and describes a solution known as "guest view casting." In their description they state, "Configuration variation over the same OS... adds additional complexity to VM semantic view reconstruction. However, the guest view casting methodology remains effective despite these differences, as shown by our evaluation..." This statement is technically true, but does not reflect the considerable difficulties in building a general approach that systematically monitors a range of guest types. The methodology is effective, but tedious, time-consuming, and requires that it be redone for every new version.

Most VM introspection research to date uses semantic reconstruction of the guest's state to acquire information (e.g., module or process lists) or to integrity check static memory structures. A more sophisticated use is to detect unauthorized tampering with dynamic memory structures, but this is a more challenging problem. Petroni's group tackles this problem by implementing a high-level language for the specification of "security predicates" [33]. The language allows them to specify constraints or invariants that indicate a security fault if violated. Our work is similar in that it allows us to monitor the guest with a high-level language. However, we believe that using C's flexibility allows developers to easily specify high-level invariants as well as work at the lower level of direct memory access.

In min-c, we have used a strategy that allows developers to create OS interface libraries automatically where symbols or source are available, or in the easiest manual fashion through specification in C. Recently machine learning (ML) approaches have begun to emerge. Payne [34] and Dolan-Gavitt [35] showed how classifiers could be trained to recognize structures from multiple versions of Windows. Kolbitsch et al. [36] have begun extracting algorithms from raw binaries. We see min-c as complementary to ML. Where source or symbols are available, automatic generation of the OS interface is always preferable because of the error rate of

ML algorithms. Where it is not, algorithms that output C data structures will allow developers to easily analyze, correct, and store their output.

VI. Conclusion

We presented min-c, a C interpreter that bridges the semantic gap problem facing applications of virtual machine introspection and volatile memory forensics, and showed its applications to VMI-specific security monitors. Like other approaches to VMI, min-c provides tamper resistance by moving security software out of a monitored VM and into the host system. Unlike other approaches, existing security monitor source code requires minimal changes because min-c makes the VM introspection invisible to the code, interpreting data structures and pointers appropriately during execution. Our approach is general within major operating system classes and automated to reduce the need for manual reverse engineering, making it attractive to both forensics and VMI applications. We believe this approach has great promise in furthering the migration of security software from monitored VMs to less vulnerable host systems and as a platform for volatile memory forensics.

References

[1] P. M. Chen, and B. D. Noble, "When virtual is better than real," in *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HOTOS '01)*, Washington, DC, USA: IEEE Computer Society, 2001, p. 133.

[2] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based 'out-of-the-box' semantic view reconstruction," in *CCS '07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, Alexandria, VA, October 2007, pp. 128– 138.

[3] B. D. Payne, M. Carbone, and W. Lee, "Secure and flexible monitoring of virtual machines," in *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)*, Miami Beach, FL, December 2007, pp. 385-397.

[4] N. Sobo. Treetop [Online]. Available: http://treetop.rubyforge.org/index.html.

[5] R. M. Stallman, and the GCC Developer Community, *Using the GNU Compiler Collection*. Boston, MA, USA: GNU Press, 2003.

[6] Microsoft Corporation. Visual studio pdb files [Online]. Available: http://msdn.microsoft.com/ en-us/library/yd4f8bd1(VS.71).aspx.

[7] S. B. Schreiber, *Undocumented Windows 2000 Secrets*. Upper Saddle River, NJ, USA: Addison-Wesley, 2001.

[8] Volatile Systems. Volatility framework [Online]. Available: https://www.volatilesystems.com/default/volatility.

[9] H. Inoue, F. Adelstein, and R. A. Joyce, "Visualization in testing a volatile memory forensics tool," To be published at the *2011 Digital Forensics Research Workshop*, Aug 2011.

[10] B. Dolan-Gavitt, B. Payne, and W. Lee, "Leveraging forensic tools for virtual machine introspection," Georgia Institute of Technology, Atlanta, GA, USA, SCS Tech. Rep. GT-CS-11-05, 2011, pp. 1-6.

[11] C. Lattner, "Llvm: An infrastructure for multi-stage optimization," Master's thesis, Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, USA, December 2002.

[12] B. Pagel, "Automated virtual machine introspection for host-based intrusion detection," Master's thesis, Engineering and Management, Air Force Institute of Technology, Wright-Patterson AFB, OH, USA, March 2009.

[13] VMware. Vmsafe partner program overview [Online]. Available: http://www.vmware.com/technical-resources/security/vmsafe/ security_technology.html.

[14] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "Revirt: enabling intrusion analysis through virtual-machine logging and replay," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, vol. 36, no. SI, pp. 211–224, 2002.

[15] Undo Software Ltd. Undodb - reversible debugging for linux [Online]. Available: http://www.undo-software.com/

[16] A. Case, L. Marziale, and G. G. Richard III, "Dynamic recreation of kernel data structures for live forensics," in *DFRWS '10: The Proceedings of the 10th Annual Digital Forensic Research Workshop (DFRWS'10)*, Portland, OR, USA, August 2010, pp. 41–47.

[17] J. Okolica, and G. L. Peterson, "Windows operating systems agnostic memory analysis," in *DFRWS '10: The Proceedings of the 10th Annual Digital Forensic Research Workshop (DFRWS'10)*, Portland, OR, USA, August 2010.

[18] VMware. VMsafe security technology [Online]. Available: http://www.vmware.com/ overview/security/vmsafe.html.

[19] T. Garfinkel, and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proceedings of the Network and Distributed Systems Security Symposium (NDSS '03)*, 2003.

[20] Mission Critical. Linux. Crash core analysis suite utility [Online]. Available: http://oss.missioncriticallinux.com/projects/crash/.

[21] M. Burdach. (2005, July 11). Digital forensics of the physical memory [Online]. Available: http://forensic.seccure.net/pdf/mburdach_digital_ forensics_of_physical_memory.pdf

[22] M. Burdach. (2005, July 9). An introduction to windows memory forensics [Online]. Available: http://forensic.seccure.net/pdf/ introduction_to_windows_memory_forensic.pdf

[23] B. Dolan-Gavitt, "The vad tree: A process-eye view of physical memory," in *Proceedings of the 8th Digital Forensic Research Workshop (DFRWS '07)*, Pittsburg, PA, USA, August 2007, pp. 62–64.

[24] N. Petroni, A. Walters, T. Fraser, and W. Arbaugh, "Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory," *Digital Investigation, the International Journal of Digital Forensics and Incident Response*, vol. 3, no. 4, pp. 197-210, December 2006.

[25] A. Schuster, "Searching for processes and threads in microsoft memory dumps," in *Proceedings of the 7th Digital Forensic Research Workshop (DFRWS '06)*, Lafayette, IN, USA, August 2006, pp. 10–16.

[26] F. D. Baiardi, and S. Sgandurra, "Building trustworthy intrusion detection through vm introspection," in *Proceedings of the 3rd IEEE International Symposium on Information Assurance and Security (IAS '07)*, Manchester, UK, August 2007, pp. 209-214.

[27] M. Bergdal, and T. A. Sorby, "Using virtual machines for integrity checking," Master's thesis, Informatics, University of Oslo, Oslo, Norway, 2007.

[28] X. Jiang, and X. Wang, "'Out-of-the-box' monitoring of VM-based high-interaction honeypots," in *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection(RAID '07)*, Queensland, Australia, September 2007, pp. 198-219.

[29] R. Jones. Virt-mem: Tools for monitoring virtual machines [Online]. Available: http://et.redhat.com/~rjones/virt-mem.

[30] N. Petroni, T. Fraser, and W. Arbaugh, "Copilot—a coprocessor-based kernel runtime integrity checker," in *Proceedings of the 13th USENIX Security Symposium (SSYM'04)*, San Diego, CA, USA, Aug 2004, pp. 179-194.

[31] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing," in *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID '08)*, Cambridge, MA, USA, pp. 1-20.

[32] G. Hoglund, "DARPA: Rootkit detection," HBGary, Tech. Rep., 2007.

[33] N. Petroni, T. Fraser, A. Walters, and W. Arbaugh, "An architecture for specification-based detection of semantic integrity violations in kernel dynamic data," in *Proceedings of the 15th USENIX Security Symposium*, Vancouver, B.C., Canada, July/August 2006, pp. 289-304.

[34] B. D. Payne, "Improving host-based computer security using secure active monitoring and memory analysis," Ph.D. dissertation, Computer Science, Georgia Institute of Technology, Atlanta, GA, USA, 2010.

[35] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin, "Robust signatures for kernel data structures," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, Chicago, IL, USA, November 2009, pp. 566-577.

[36] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda, "Inspector gadget: Automated extraction of proprietary gadgets from malware binaries," in *Proceedings of the 31st IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2010, pp. 29-44.