TEMU: Binary Code Analysis via Whole-System Layered Annotative Execution

Heng Yin
Dawn Song

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-3
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-3.html

January 11, 2010

Copyright © 2010, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

TEMU: Binary Code Analysis via Whole-System Layered Annotative Execution

Heng Yin
Dept. of Electrical Eng. & Comp. Science
Syracuse University
Syracuse, NY 13244
heyin@syr.edu

Dawn Song
Computer Science Division
UC Berkeley
Berkeley, CA 94720
dawnsong@cs.berkeley.edu

ABSTRACT

Dynamic binary analysis has demonstrated its strength in solving a wide-spectrum of computer security problems, such as malware analysis, protocol reverse engineering, vulnerability detection, diagnosis, and defense, software testing, etc. An extensible platform for dynamic binary analysis provides a foundation for solving these problems. To enable a variety of applications, we explore a unique design space. We aim to provide a whole-system view, take an external approach, facilitate fine-grained instrumentation, and have sufficient efficiency. These design goals bring about a new architecture, namely *whole-system out-of-the-box fine-grained dynamic binary analysis*. To further facilitate fine-grained dynamic binary analysis, we propose *layered annotative execution* as a core technique, which incorporates shadow flag analysis, taint analysis, and symbolic execution. We have implemented this new architecture and the core technique in an analysis platform called *TEMU*. Because of its extensibility and versatility, TEMU has enabled and fostered a handful of research projects.

1. INTRODUCTION

Dynamic binary analysis has demonstrated its strength in many research problems, such as malware analysis, protocol reverse engineering, vulnerability signature generation, software testing, profiling and performance optimization, etc. An extensible platform for dynamic binary analysis serves as a foundation for solving these problems, and therefore becomes a critical building block.

In this paper, we present an extensible platform that addresses the common challenges and requirements in dynamic binary analysis, and thus greatly facilitates building various custom analysis techniques on top of it. First of all, we often need a whole-system view, including the OS kernel and all the application running on the system. Such a whole-system view enables us to analyze the activities happening in the OS kernel (such as kernel malware and kernel vulnerabilities) and interactions between multiple processes. In addition, we prefer an *external* approach. That is, we want to perform analysis completely outside the execution environment. This out-of-the-box approach provides excellent isolation and good transparency. It makes it more difficult for malware to detect the presence of the analysis environment and interfere with analysis results. Moreover, in many cases, we need to perform deep analysis by inspecting in a fine-grained manner (i.e., at the instruction level). Finally, the analysis overhead should be acceptable.

To address these challenges, we propose a new architecture for dynamic binary analysis, called *whole-system out-of-the-box fine-grained dynamic binary analysis*. The basic idea is to run an entire operating system (including common applications) inside a whole-system emulator, and execute the binary code of interest in this emulated environment. During execution of the binary code, we monitor and analyze its behavior at the instruction level, completely from the emulator. We propose and implement a core technique, namely *layered annotative execution*, as a Swiss army knife, to fine-grained binary code analysis. Essentially, during the execution of each instruction in the emulated environment, depending on the instruction semantics and the analysis purpose, we can annotate certain memory locations or CPU registers or update existing annotations. This is a layered approach, because we can place extra analysis processes on top of the existing analysis to extract more insightful results. We implement the new architecture and the core technique into a generic dynamic binary analysis platform, codenamed *TEMU*. It is based on an open-source whole-system emulator, QEMU [2].

TEMU has been widely used for many research projects in numerous categories, as listed below:

- **Malware analysis.** The objective of malware analysis is to automatically extract important insights of an unknown malicious program. Panorama [42] detects and analyzes a myriad of malware such as keylogger, spyware, rootkits and backdoors, which access sensitive information in abnormal ways. Renovo [24] is a generic unpacker, which extracts original code and data from packed malicious binaries. HookFinder [41] identifies hooks installed by malware and provides insightful knowledge about hooking mechanisms. BitScope [6] uncovers potentially hidden functionality of malware, and provides semantic understanding of dependencies between inputs and outputs.
- **Protocol reverse engineering.** Protocol reverse engineering aims to extract the knowledge of application-level protocol format and semantics directly from its

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Submitted to VEE '10, Pittsburgh, PA, USA
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

implementation. Polyglot [12] extracts the format and semantics of incoming protocol messages. Dispatcher [11] supersedes Polyglot in the following aspects: 1) dealing with encrypted messages; 2) analyzing bidirectional protocol messages; and 3) extracting more semantic information.

- **Vulnerability detection, diagnosis, and defense.** Software vulnerabilities (e.g., buffer overflows) can be exploited by attackers to compromise millions of machines in hours or even minutes. TEMU has been used to automatically detect exploits to previously unknown vulnerabilities [32], automatically generate filters to protect vulnerable hosts [8, 31], and used to demonstrate that automatically generating exploits from security patches is possible [9].

In summary, we have made the following contributions:

- We systematically study the common requirements and challenges in a wide-spectrum of security applications of dynamic binary analysis. In particular, whole-system view, external approach, and fine granularity are important for many applications.
- We have proposed a new analysis architecture, *whole-system out-of-the-box fine-grained dynamic binary analysis*, to address these requirements and challenges.
- We have devised a core technique for fine-grained binary analysis, namely *layered annotative execution*, as a versatile solution for various analysis needs.
- We have designed and implemented an extensible platform, *TEMU*, to realize this new architecture and core technique.
- We have evaluated the capabilities, extensibility, and efficiency of this platform by demonstrating that we can build various tools easily on top of it to enable a suite of security applications.

2. SYSTEM OVERVIEW

2.1 Design Goals

We aim to develop an analysis platform that provides sufficient support for a wide-spectrum of analysis purposes. We have identified the following design goals, which no existing analysis frameworks have addressed systematically.

Whole-system view. Having a complete view of the execution environment (including the OS kernel and all the applications) is critical for many analysis problems. Malware often infiltrates into the OS kernel, injects code into another process, and interacts with multiple processes. Zero-day worm exploits sometimes involve multiple processes and the OS kernel [15]. Without a whole-system view, we are unable to obtain a complete picture of these malicious activities. However, many analysis platforms (e.g., Valgrind [29], DynamoRIO [5], Pin [25]) only provide a local view (i.e., a view of a single user-mode process).

External monitoring approach. We prefer to take an *external* approach versus an *internal* monitoring approach. In an external approach, no modification is made in the execution environment to be analyzed, and the execution is observed and instrumented completely from outside. In contrast, an internal monitoring approach, taken by Valgrind [29], DynamoRIO [5], and Pin [25], places the analysis tool into the same execution environment to be analyzed. Therefore, the system state and memory layout are perturbed. The external monitoring approach provides excellent isolation and better transparency. As a result, it makes it more difficult for malware to detect the presence of the analysis environment and interfere with analysis results.

OS awareness. A central challenge brought by the whole-system view and external monitoring approach is the semantic gap. That is, we can only see the hardware-level view of the analyzed system, such as the states of cpu registers, physical memory, and IO devices. However, we need an OS-level view to get meaningful analysis results. For example, we need to know what process is currently running and what module an instruction comes from. Therefore, we need techniques that can be used to extract the OS-level semantics from the analyzed system.

Support for fine-grained analysis. Many analysis problems require fine-grained instrumentation (i.e., at instruction level) on binary code. Traditional debugging techniques (such as hardware breakpoint and single stepping) incur significant performance overhead, which is often hundreds of times slowdown or even more. Dynamic translation is a technique that translates code at runtime and caches the translated code. Consequently, dynamic translation provides a more efficient foundation for fine-grained instrumentation. Tools like Valgrind [29], DynamoRIO [5], Pin [25], and QEMU [2] take this approach to facilitate fine-grained instrumentation.

2.2 Architecture

To address the above design goals, we propose a new architecture, *whole-system out-of-the-box fine-grained dynamic binary analysis*. The basic idea is to run an entire operating system (including common applications) inside a whole-system emulator, execute the binary code of interest in this emulated environment, and then observe and analyze the behaviors of this binary code from the emulator. Figure 1 illustrates this new analysis architecture.

This new architecture is able to address all the above design goals. First of all, such a whole-system emulator emulates a full system, including CPU, physical memory, and IO devices. Therefore, it has a complete view of the emulated execution environment. In addition, performing analysis within this whole-system emulator follows the external monitoring approach. That is, we completely observe the activities of the emulated system from outside. On one hand, this whole-system emulation approach provides excellent isolation between the emulated guest system and the emulator. On the other hand, we are aware that whole-system emulation can still be detected [19, 26, 34]. We leave this transparency issue as future work. To bridge the semantic gap, we devise a set of mechanisms that can extract the OS-level semantics from the emulated guest system. We implement these mechanisms in the *Semantic Extractor*. Finally, this whole-system emulator makes use of dynamic translation techniques to emulate the guest system execution, and therefore facilitate fine-grained instrumentation.

We make a key observation that many analysis techniques (such as shadow flag analysis [24], dynamic taint analysis [12, 32, 41, 42], and symbolic execution [6, 11, 30]) need to

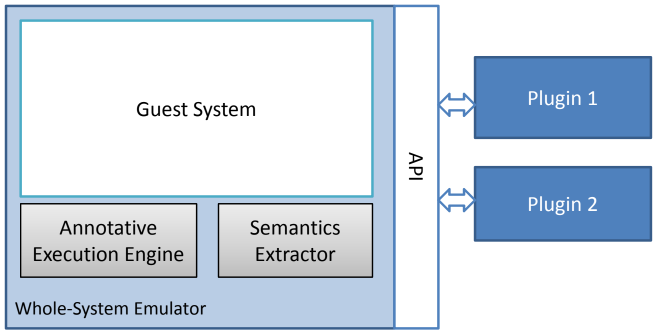


Figure 1: Architecture of TEMU

annotate a memory location or CPU register based on the semantics of the currently executed instruction, or update the existing annotations. Therefore, we propose a unified analysis technique, *annotative execution*, to summarize and generalize these specific techniques. Furthermore, a more advanced analysis technique can be built on top of a more basic analysis technique. For instance, dynamic taint analysis can be built on top of shadow flag analysis, and symbolic execution can be built on top of dynamic taint analysis. Therefore, this annotative execution has an inherently layered structure. We implemented this *layered annotative execution* in the *Annotative Execution Engine*.

To facilitate implementing custom analysis, we have defined a clear interface between the platform and a plugin. Such an interface can hide unnecessary details from users and reuse common functionalities.

We have implemented a new analysis platform, *TEMU*, in Linux, based on an open-source whole-system emulator, QEMU [2]. It has become the dynamic analysis component in the BitBlaze binary analysis infrastructure [4, 37]. At the time of writing, TEMU can be used to analyze binary code in Windows 2000, Windows XP, and Linux systems. We describe these three components in Section 3, 4 and 5 respectively.

3. SEMANTICS EXTRACTOR

The semantics extractor is responsible for extracting OS-level semantics information of the emulated system, including process, module, thread, symbol information, and execution context. The mechanisms presented below are extensions to the technique of *virtual machine introspection*, which was first proposed for host intrusion detection [22] and later extended for malware detection [40].

3.1 Process and Module Information

For the current execution instruction, we need to know which process, thread and module this instruction comes from. In some cases, instructions may be dynamically generated and executed on the heap.

Maintaining a mapping between addresses in memory and modules requires information from the guest operating system. We use two different approaches to extract process and module information for Linux and Windows.

For Linux, we can directly read process and module information from outside, because we know the relevant kernel data structures, and the addresses of relevant symbols are also exported in the `system.map` file. In order to maintain the process and module information during execution, we hook several kernel functions, such as `do_fork` and `do_exec`.

For Windows, we do not have access to the kernel source code. Although some kernel data structures have been reverse engineered for some versions of Windows, this is not a robust way to extract OS-level semantics across different versions and service packs. Instead, we have developed a kernel module for this purpose. We load this module into the guest operating system to collect the updated memory map information. This kernel module registers two callback routines. The first callback routine is invoked whenever a process is created or deleted. The second callback routine is called whenever a new module is loaded and gathers the address range in the virtual memory that the new module occupies. In addition, the kernel module obtains the value of the CR3 register for each process. As the CR3 register contains the physical address of the page table of the current process, it is different (and unique) for each process. All the information described above is passed on to TEMU through a predefined I/O port. This is a suboptimal solution currently, in that we favor portability over transparency. We leave a better solution to this problem as future work.

3.2 Thread Information

We obtain the current thread ID, which is important for analyzing multi-threaded programs. In Windows, the data structure for the current thread is mapped into a well-known virtual address, so we can obtain this information directly. For Linux, we can also obtain the thread information by a similar approach. However, the current implementation of TEMU only gets thread information from Windows for now, and will support Linux in the future.

3.3 Symbol Information

Given a binary module, we parse its header information in memory and extract the exported symbol names and offsets. After we obtain the locations of all modules, we can determine the absolute address of each symbol by adding the base address of the module and its offset. This feature is very useful, because all APIs in user libraries and kernel APIs are exported by their hosting modules. The symbol information conveys important semantics information, because from a function name, we are able to determine what purpose this function is used for, what input arguments it takes, and what output arguments and return value it generates. Moreover, the symbol information makes it more convenient to hook a function—instead of giving the actual address of a function, we can specify its module name and function name. Then TEMU will automatically map the actual address of the function for the user. In the current implementation, TEMU is able to parse memory images of PE and ELF binary modules, and thus supports both Windows and Linux.

3.4 Execution Context

With the knowledge of processes and modules, we are able to tell which process or module performs a certain behavior directly. However, if this behavior happens in library or system code, it is actually performed on behalf of the program that invokes the functionality in system or library code. Therefore, we need to determine if a certain behavior is performed under the execution context of the program to be analyzed.

External function calls. First, we consider the case where the program under analysis makes an external function call, whose function body is located in another code module (i.e., library). We use the following observation: Whenever a program makes a function call, the value of the stack pointer at the time of the function call must be greater than the value of the stack pointer at the time when a certain behavior is performed in that external function call. This is because one or more stack frames have to be pushed onto the stack when making function calls, and the stack grows toward smaller addresses on the x86 architecture.

Based on our observation, we use the following approach to identify the execution in an external function: Whenever the execution jumps into the code under analysis, we record the current value of the stack pointer, together with the current thread identifier. When execution jumps out of this code region, we check whether there is a recorded stack pointer for the current thread identifier, and if so, whether this value is smaller than the current stack pointer. If this is the case, we remove the record as the code is not on the stack anymore. Whenever an interesting behavior is observed, we check whether there is a recorded stack pointer under the current thread identifier. If so, we consider this behavior to be performed on behalf of the code under analysis, because this means that the code under analysis is on the call stack.

Software interrupts. The program under analysis may make a software interrupt (e.g, `int 80` for system call in x86), and then the kernel will service this software interrupt. Therefore, we need to attribute the behavior happening inside the software interrupt to the program under analysis. In the whole-system emulator, we can easily observe all the software interrupts. Then we can attribute the execution during this software interrupt under the same thread context to the program's context.

Fast system calls. The program under analysis may make a fast system call (i.e., `sysenter` in x86), and there is no software interrupt happening for this fast version. To associate behaviors happening in a system call with the program that executes `sysenter`, we instrument `sysenter` and `sysexit` instructions, and attribute the execution in the same thread context between `sysenter` and `sysexit` to the program under analysis.

4. ANNOTATIVE EXECUTION ENGINE

We propose a generic technique for dynamic binary code analysis, *layered annotative execution*. During the execution of each instruction, depending on the instruction semantics, we can *annotate* the operands of this instruction, or propagate annotations from source operands to destination operands, or update the existing annotations.

We can perform annotative execution in a variety of ways. The most basic analysis is *shadow flag analysis*, in which we may simply annotate certain memory locations or registers to be *dirty* or *clean*. A more advanced analysis is *dynamic taint analysis*, in which we not only annotate certain memory locations and registers to be *tainted*, but also keep track of taint propagation. The most advanced analysis is *symbolic execution*, in which we not only mark certain inputs (i.e, memory locations or registers) as tainted, but assign a meaningful symbol to these inputs. Then during taint propagation, we associate symbolic expressions to the tainted memory locations and registers. These symbolic expressions indicate how these variables are calculated from the symbolic inputs. This is a layered approach: one analysis mechanism is built on top of another to perform more advanced analysis, as illustrated in Figure 2.

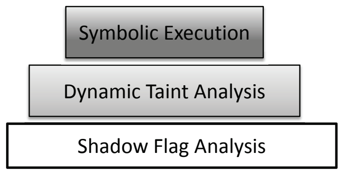


Figure 2: A Layered Approach for Annotative Execution

Shadow memory. We use a *shadow memory* to store and manage the annotations of each byte of the physical memory and CPU registers and flags. To support tracking memory being swapped in and out, we also have shadow memory for the hard disks. With the shadow memory for the hard disks, the system can continue to track the annotations that have been swapped out.

```
uint64_t regs_bitmap = 0; //bitmap for CPU registers
uint8_t *regs_records; //annotations for CPU registers

typedef struct _tpage_entry {
  uint64_t bitmap;    //bitmap for 64-byte memory chunk
  uint8_t records[0]; //byte array for 64 annotations
} tpage_entry_t;
tpage_entry_t **tpage_table; //page table for main memory

typedef struct disk_record
  uint64_t index; //index for a 64-byte disk chunk
  uint64_t bitmap; //bitmap for the 64-byte disk chunk
  struct list_head link; //linked list entry
  uint8_t records[0]; //byte array for 64 annotations
 disk_record_t;
struct list_head disk_hashtable[1024]; //hash table for disk
```

Figure 3: Data structures for shadow memory

The code snippets in Figure 3 illustrate how TEMU maintains the shadow memory. The shadow memory for CPU registers is straightforward, because there are only a small number of registers. A 64-bit integer `regs_bitmap` is big enough to shadow each byte of 8 general-purpose registers in 32-bit x86 architecture. `regs_records` is a byte array to store annotations for registers. The type of annotation record should be defined by the plugin, and is opaque to the platform. Hence, TEMU just treats an annotation record as a sequence of bytes, and only needs to know the size of an annotation record. Once the size of annotation record is provided by the plugin, `regs_records` will be allocated with appropriate size.

To ensure efficient memory usage, the shadow memory for main memory is organized as a one-level page table. Each page entry responds to 64 bytes in main memory. If an entry is `NULL`, it means no annotations associated with any of the 64 bytes in it. Otherwise, the `bitmap` field indicates which of 64 bytes have annotations. The `records` is the byte array for storing 64 annotation records. Similarly, the `records` field is defined as a zero-length array, and will be allocated with appropriate size when the size of annotation record is provided by the plugin.

The shadow memory for the hard disk is managed differently. Because the size of a hard disk is several orders of magnitude larger than main memory, a 1-level page table will not be space efficient. A multiple-level page table is more space efficient at the price of page lookup speed. Considering that in practice, a very small amount of disk data are annotated (for page swapping and small data files), we use a hash table. The disk space is divided into 64-byte chunks, and the index to a disk chunk is used to look up the hash table. The size of hash table is independent of the disk size, and lookups in this hash table are efficient when only a small number of disk chunks are annotated.

Code instrumentation. We instrument the execution of the guest system to enable annotative execution. We also implant callbacks to notify the plugin of various events, as shown in Section 5.

QEMU uses dynamic translation techniques to emulate a guest system. That is, when a block of code (i.e, a code stream with a single entry and a single exit) from the guest system is executed for the first time, it is translated into statements in an intermediate form, and then the code executed on the host system is generated from these intermediate statements. The translated code is stored in the code cache, such that the code translation overhead can be amortized over subsequent executions of the same code block.

The code instrumentation is intermingled with a dynamic translation process. In general, there are two ways to insert instrumentation code: 1) inline the instrumenting instructions or statements directly into the translated code, as Memcheck [28] and Argos [33] do; and 2) insert function calls, and the actual instrumentation tasks are implemented in these functions. The advantage of the first approach is efficiency, because some redundant operations can be optimized away during the code emission phase. Its disadvantage is lack of capability and extensibility. It is cumbersome and sometimes infeasible to implement comprehensive functionality for instrumentation. The second approach is on the contrary: less efficiency but greater capability. Therefore, we take the second approach, in favor of capability over efficiency. We plan to investigate performance optimization in future work.

In Appendix A, we use a concrete example to walk through this code instrumentation process in more detail.

4.1 Shadow Flag Analysis

Shadow flag analysis is the most basic analysis in this layered architecture. Basically, depending on the execution context and the semantics of the current instruction, we can decide whether to create an annotation for a memory location or register. Later, we can check or change this annotation. Compared to the other more advanced analysis techniques, shadow flag analysis is the most efficient.

4.2 Taint Analysis

Our dynamic taint analysis is similar in spirit to a number of previous systems [13,14,16,32,38]. In comparison, our design and implementation is the most complete. For example, previous approaches either operate on a single process only [14,32,38], or they cannot deal with memory swapping and disks [13,16].

Taint source. A plugin is responsible for introducing taint sources into the system. TEMU supports taint input from hardware, such as the keyboard, network interface, and hard disk. TEMU also supports tainting a memory region (e.g. the output of a function call, or a data structure in a specific application or the OS kernel).

Basic propagation policy. After a data source is tainted, we need to monitor each CPU instruction and DMA operation that manipulates this data in order to determine how the taint propagates. For data movement instructions and DMA operations, the destination will be tainted if and only if the source is tainted. For arithmetic instructions, the result will be tainted if and only if any byte of the operands is tainted. We also handle the following special situations.

Constant function. Some instructions or instruction sequences always produce the same results, independent of the values of their operands. A good example is the instruction "`xor %eax, %eax`" that commonly appears in IA-32 programs as a compiler idiom. After executing this instruction, the value of `eax` is always zero, regardless of its original value. We recognize a number of such special cases and untaint the result.

Table lookup. A tainted input may be used as an index to access an entry of a table. The taint propagation policy above will not propagate taint to the destination, because the value that is actually read is untainted. Unfortunately, such table lookup operations appear frequently, such as for Unicode/ASCII conversion in Windows. Thus, TEMU has an option to propagate taint through table lookups: if any byte used to calculate the address of a memory locations is tainted, then, the result of a memory read using this address is tainted as well. The plugin has capability to enable or disable this option.

Logic and bit shifting. We need to take care of these operations with more precision. For example, for "`and $0xff, %eax`", if all of the 4 bytes of `eax` is tainted before this instruction, then only the lowest byte of `eax` should be tainted after the execution of this instruction. Similarly, for "`shr $24, %eax`", only the lowest byte of `eax` should be tainted if `eax` the highest byte is tainted originally. We cover these special instructions to track taint more precisely.

Tracking multiple taint labels. For many analysis purposes, it is often necessary to introduce and track multiple taint sources simultaneously. When an instruction (e.g., `add`) has multiple tainted source operands, the destination operand (often) needs to be marked as tainted from the multiple taint labels. How to maintain multiple labels is in fact application-specific. The plugin may choose to pick one label, maintain up to a number of labels, or maintain all of the labels. Thus, TEMU asks the plugin to handle this situation when multiple taint labels converge.

4.3 Symbolic Execution

Symbolic execution gives abstract interpretation of how certain values are processed on both the data plane and the control plane. On the data plane, symbolic execution allows registers and memory locations to contain symbolic expressions in addition to concrete values. Thus, a value in a register may be an expression such as `X + Y` where `X` and `Y` are symbolic variables. Consider a small program in Figure 4. After execution, we produce a symbolic expression for `mem[10]`, which is `mem[10] = y*3+5`. This symbolic expression abstractly interprets how the content in this memory location is calculated from the relevant symbolic inputs on the data plane.

On the control plane, symbolic execution generates a path predicate, describing the constraints on the symbolic inputs

```
L1: z = 10;
L2: x = 2;
L3: x = y*3;
L4: z = x+4;
L5: k = z+1;
L6: if(z<10)
L7:   mem[10] = k;
```

Figure 4: A Simple Symbolic Program

needing to be satisfied for the program execution to go down that path. In the above example, the if statement $z < 10$ has to be true for the `mem[10]` to be assigned a new value. The symbolic execution can give us a path predicate `y < 2`, which abstractly describes what condition has to be satisfied in order to perform the assignment operation on L7.

When certain conditions are not satisfied, behaviors depending on these conditions will not be exhibited. In the above example, if the actual value of `y` is 3, then the if statement `z<10` will not be true, and the operation on L7 will not be executed. To uncover the hidden behaviors, for each control flow decision that depends on symbolic inputs, we will determine which branches are feasible and try to explore all the feasible execution paths. More precisely, for each branch, we extract a symbolic expression as the path predicate, and use a theorem prover to determine if the path predicate can be *true*. In the above example, we will be able to explore both branches for the if statement on L6, because we determine the path predicate can be either *true* or *false*. Thus, we will be able to uncover the memory assignment on L7.

During symbolic execution, for each instruction, we need to determine if it should be executed symbolically. If so, we enqueue this instruction and its operands into the symbolic machine. As a consequence, the instructions and states in the symbolic machine form a symbolic program. Then if we want to query the symbolic expression and path predicate of a symbol, we extract formulas from the symbolic program. In addition, whenever a control flow decision is dependent on a symbolic variable, we attempt to explore all feasible directions.

Generate symbolic program. An instruction can be executed concretely iff all operands of the instruction are concrete. Thus, deciding whether an instruction should be executed concretely or symbolically requires information about which data in the system is concrete and which is symbolic. Recall that the shadow memory associated with registers and memory indicates the status of each byte. A symbolic byte is marked as tainted. Thus, to determine if an instruction needs to be executed symbolically, we just need to check if any of its operands is tainted. If so, we perform symbolic execution, and mark the destination operand as tainted, just like normal taint propagation. Otherwise, we execute this instruction concretely.

Mixed symbolic and concrete execution means that many instructions will be executed concretely and never be executed on the symbolic machine. Therefore, if an instruction to be symbolically executed has any concrete operands, we must update those concrete values inside the symbolic machine.

Ideally, during symbolic execution, we would like to generate symbolic expressions and path predicates on the fly. However, this naive approach would incur unacceptable performance overhead at runtime. To optimize the performance, we perform "lazy" symbolic execution. Its basic idea is to quickly perform as few operations as possible to guarantee fast runtime performance, and maintain enough information for post analysis. Specifically, for each instruction that needs to be executed symbolically, we enqueue that instruction, along with the relevant machine states (including all operands and other related memory and register states) into our symbolic machine. Then we quickly mark the destination operand as symbolic by checking the source operands. This strategy enables fast runtime performance. As a consequence, the instructions and states in the symbolic machine form a symbolic program, just like the one in Figure 4.

Extract symbolic formulas. We take the following steps to extract a symbolic formula for a symbol from the symbolic program. First, we perform dynamic slicing on the symbolic program. This step removes the instructions that the symbol does not depend upon. After this step, the symbolic program will be reduced tremendously. Then we generate one expression by substituting intermediate symbols with their right-hand-side expressions. Finally, we perform constant folding to further simplify the expression. Still using the program in Figure 4 as an example, to get the symbolic expression for `mem[10]`, we perform dynamic slicing first. It would remove the instructions on L1 and L2. Then we perform symbol substitution, and we get a formula like below:

```
mem[10] = y*3+1+4
```

Then we perform constant folding on it, and finally get:

```
mem[10] = y*3+5
```

Explore multiple execution paths. When executing a conditional jump instruction that depends on a symbolic condition, we attempt to explore all feasible paths. To determine if a path is feasible, we generate the path predicate for that path, and ask the Solver if this path predicate is satisfiable. The Solver is a theorem prover or decision procedure, which performs reasoning on symbolic formulas. TEMU is extensible; we can plug in any Solver appropriate, and our system thus can automatically benefit from any new progress on decision procedures, etc. Currently in our implementation, we use STP as the Solver [20, 21].

A satisfiable path predicate means a feasible path. The plugin needs to decide which feasible direction needs to be explored now. It needs an algorithm to prioritize the paths in the malicious code. For example, it can use breadth-first search, depth-first search, and other strategies.

Once we decide which direction to explore, we save the state of the emulated system, and then make the system execution go to that direction by changing the EIP register. Later, if we want to explore the other direction, we can simply restore the state and start execution from that point. More specifically, the saved state includes the states of whole emulated machine (such as registers, memory, and I/O devices), the state of shadow memory in TEMU, and the symbolic program. The size of this entire state can be large. We can employ various compression techniques to reduce the size. For example, we can save the relative state changes to an initial state, instead of the absolute state. Then we can perform common compression methods on the relative state to further reduce its size.

The functionality of state saving and restoring enables a distributed architecture for binary code analysis. It may be still time-consuming to analyze a complex and big binary program, in terms of the number of branches that depend upon symbolic inputs. A centralized controller may disseminate different saved states to multiple working nodes, such that they can explore multiple different execution paths in parallel. This architecture would significantly reduce the overall analysis time.

Moser *et al.* also built system that is capable of exploring multiple execution path for malware analysis [27]. In comparison, our implementation is more comprehensive. First, TEMU maintains path predicates with bit-level accuracy and can handle non-linear path constraints, whereas their system can only handle linear constraints. Second, their system saves and restores states for a specific process, assuming malware is only within one process, while our system handles whole-system states and thus can cope with malware that involves kernel code and multiple processes.

5. TEMU APIS

In order for users to make use of the functionalities provided by TEMU, we define a set of functions and callbacks. By using these interfaces, users can implement their own plugins and load them into TEMU at runtime to perform custom analysis tasks. Currently, TEMU provides the following functionalities:

- Switch between emulation and virtualization mode. With support of a kernel accelerator, QEMU can be configured to run under virtualization mode, which is nearly as efficient as native execution. We enhance this feature to enable or disable virtualization mode at runtime. For example, we can boot up the guest system under virtualization mode, and switch to emulation mode just before analyzing a program.
- Query and set value of a memory cell or a register.
- Query and set annotation of a memory cell or register.
- Register and remove a function hook. We can hook a function by either its entry point or its name (if the function name is exported). We can hook both the entry and exit of a function call. To determine the exit point, we obtain the return address from the call stack on the function entry. We use the thread identifier to distinguish different call instances, so function hooks in multi-threaded environment are properly handled.
- Query OS-level semantics information. We can query information about processes, modules, threads, execution contexts, etc.
- Save and load the guest system state. This interface helps to switch between different machine states for more efficient analysis. QEMU can save and load a virtual machine state. We extend this feature to save annotations and other analysis states into the virtual machine state, and then load it back. This extended feature facilitates multiple path exploration, because we can save a state for a specific branch point and explore one path, and then load this state to explore the other path without restarting the program execution.

TEMU defines callbacks for a set of events, including (1) the entry and exit of a basic block; (2) the entry and exit of an instruction; (3) propagation of annotations; (4) memory read and write; (5) register read and write; (6) interrupts and exceptions; (7) events of IO devices such as network and disk inputs and outputs; (8) OS-level events, such as process creation and exit, and module loading and unloading; and so on.

6. EVALUATION

In this section, we evaluate this analysis platform with respect to its capabilities, extensibility and ease of application development. That is, we want to see on top of TEMU: 1) how different kinds of analysis tools can be built; and 2) how easy these tools can be implemented.

To evaluate these metrics, we present four malware analysis tools that are built on top of TEMU. They are *Renovo* [24] (a generic unpacker), *Panorama* [42] (an information flow tracker), *HookFinder* [41] (a hook analyzer), and *MineSweeper* [7] (a hidden behavior explorer).

6.1 Renovo: Generic Unpacker

Malware often employs various code obfuscation techniques. One common technique is code packing, which transforms a program into a packed executable by compressing or encrypting the original code and data into packed data. When this packed executable gets executed, the embedded unpacking routine recovers the original code and data and transfers the execution to the original entry point. Renovo [24] captures an intrinsic nature of code packing behavior. That is, independent of specific packing techniques, original code and data regions belong to memory regions that are modified during the unpacking process, and later the instruction pointer jumps into some of these modified regions to execute unpacked code.

Renovo makes use of shadow flag analysis functionality provided by TEMU. Renovo runs the program to be unpacked in TEMU. When the program executes a memory write instruction, it marks the corresponding destination memory region as *dirty*, which means it is newly generated. During the program execution, when it sees the instruction pointer jump to one of these newly-generated memory regions, Renovo determines that this is the unpacked code. Apparently, this tool needs OS-level semantics provided by TEMU to know whether the current execution is under the context of the program to be unpacked.

Experimental results. Both synthetic and real-world samples were used to evaluate the effectiveness and efficiency of Renovo. The results from Renovo were also compared with those from other two unpackers, UUnP [17] (a plugin of IDA Pro) and PolyUnpack [35] (an unpacker based on OllyDebug).

	Renovo	UUnP	PolyUnpack
Unpacked	12	6	6
Avg. Time (sec.)	14+30	26	656

Table 1: Unpacking Synthetic Samples.

Synthetic samples were generated by applying 15 different packers on Microsoft Notepad. Table 1 summarizes the results. While Renovo can unpack the majority of synthetic samples (12 out of 15), UUnP and PolyUnpack can only unpack 6 out of 15 samples.

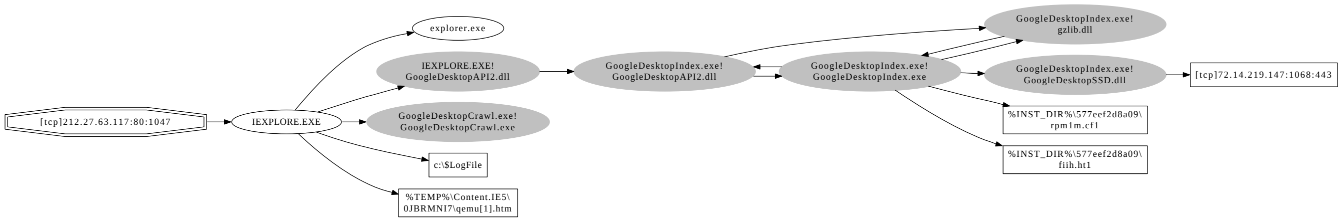

[tcp]212.27.63.117:80:1047
IEXPLORE.EXE
explorer.exe
IEXPLORE.EXE! GoogleDesktopAPI2.dll
GoogleDesktopCrawl.exe! GoogleDesktopCrawl.exe
c:\$LogFile
%TEMP%\Content.IE5\ 0JBRMNI7\qemu[1].htm
GoogleDesktopIndex.exe! GoogleDesktopAPI2.dll
GoogleDesktopIndex.exe! GoogleDesktopIndex.exe
GoogleDesktopIndex.exe! gzlib.dll
GoogleDesktopIndex.exe! GoogleDesktopSSD.dll
[tcp]72.14.219.147:1068:443
%INST_DIR%\577eef2d8a09\ rpm1m.cf1
%INST_DIR%\577eef2d8a09\ fiih.ht1

Figure 5: A Taint Graph for Google Desktop

	Renovo	UUnP	PolyUnpack
Extracted results	366	186	171
IRC pattern found	363	176	86
Avg. time (sec.)	10.9+30	15.7	365.8

Table 2: Unpacking Real-world Malware Samples.

Table 2 shows the results against 374 real-world packed malware samples. Similarly, Renovo is able to unpack the vast majority of them, whereas UUnP and PolyUnpack can only handle less than half. In terms of efficiency, considering that Renovo needs approximately 30 seconds for system to boot up, Renovo is in fact the most efficient among these three tools. In particular, PolyUnpack, as a fine-grained analysis tool on top of a debugger, is significantly slower than Renovo based on TEMU. Therefore, it demonstrates the efficiency of TEMU, due to dynamic binary translation.

6.2 Panorama: Information Flow Tracker

A wide-spectrum of malware categories, including spyware, keyloggers, network sniffers, stealth backdoors, and rootkits, share similar fundamental characteristics. They tend to access, tamper with, and leak sensitive information in abnormal ways. Panorama is built on top of TEMU to capture this abnormal information access behavior.

Panorama [42] utilizes the dynamic taint analysis functionality implemented in TEMU to capture information access behaviors. Several unique features in this taint analysis are critical for this tool. First, our taint analysis is performed on the whole system, and therefore can capture system-wide information flows. Second, it is customizable: the plugin can define its own data structure for taint record and track multiple taint sources simultaneously. Finally, the OS-level semantics provided by TEMU are used to lift up the hardware-level taint propagation events to OS level, and therefore to generate OS-level taint propagation graphs.

Experimental results. Google Desktop was chosen for a case study. Google Desktop would send sensitive user information such as the local search index files back to Google's servers in certain configuration settings [23]. Within 30 minutes, Panorama was able to automatically capture this information leakage and present insightful results in the form of OS-level taint graphs. Figure 5 illustrates a representative taint graph. From this taint graph, we can see that a webpage first was received by the Internet Explorer, and then a component from Google Desktop (`GoogleDesktopAPI2.dll`) obtained the web page and passed it over to a stand-alone program also from Google Desktop (`GoogleDesktopIndex.exe`). This stand-alone process further processed this information and saved it into two data files (`rpm1m.cf1` and `fiih.ht1`) in its local installation directory. Eventually it sent some information derived from the web page to a remote Google server (72.14.219.147) through an HTTPS connection. This taint graph demonstrates the necessity of having a whole-system view, because this information leakage involved multiple processes and components.

6.3 HookFinder: Hook Analyzer

One important malware attacking vector is its hooking mechanism. Malware implants hooks for many different purposes. Spyware may implant hooks to get notified of the arrival of new sensitive data. Rootkits may implant hooks to intercept and tamper with critical system information to conceal their presence in the system. A stealth backdoor may also place hooks on the network stack to establish a stealthy communication channel with remote attackers. Existing hook detection tools detect hooks by checking known memory regions for suspicious entries. To evade detection, malware tends to install hooks in previously unknown memory regions.

HookFinder [41] aims to automatically detect malware's hooking behaviors (especially previous unseen ones), and extract insightful knowledge about hooking mechanisms. It captures an inherent property of hooking behavior: a hook is one of the impacts (i.e., state changes in the execution environment caused by malware) that redirects the system control flow back into the malicious code. To characterize the impacts of malware, HookFinder performs *fine-grained impact analysis*, which is a variant of dynamic taint analysis. This variant was easily implemented on top of TEMU. Thus, it demonstrates that TEMU is flexible and extensible.

Sample	Size	Runtime		Hooks
		Online	Offline	
Troj/Keylogg-LF	64KB	6m	9m	2
Troj/Thief	334KB	4m	3s	1
AFXRootkit	24KB	6m	33m	4
CFSD	28KB	4m	2m	5
Sony Rootkit	5.6KB	4m	2s	4
Vanquish	110KB	5m	12m	11
Hacker Defender	96KB	5m	27m	4
Uay Backdoor	212KB	4m	25s	5

Table 3: Hook Detection and Analysis Results

Experimental results. Eight real-world malware samples were used to evaluate the effectiveness and efficiency of HookFinder. Five of these samples have presence in the kernel space. Therefore, it is critical to have a whole-system view, in order to analyze these malicious behaviors in the kernel space. The runtime breaks down into an online detection phase and offline analysis phase. In the online detection phase, a malware sample was run in TEMU, hooking behaviors were identified, and an impact trace was generated.

In the offline analysis phase, backward dependency analysis was performed on the impact trace, and hook graphs were generated. Within 6 minutes, HookFinder was able to detect the hooking behaviors of all the samples during the online detection phase. Then in up to 27 minutes, HookFinder extracted hook graphs from the impact trace. It is worth noting that HookFinder was able to detect and analyze the new hooking technique employed by the Uay backdoor.

6.4 MineSweeper: Hidden Behavior Explorer

Malware often contains hidden behavior which is only activated when properly triggered. For example, many viruses attack their host systems on specific dates; some keyloggers only record keystrokes to files when the application window name contains certain keywords; some distributed denial-of-service tools only start launching attacks when receiving certain network commands. Detecting these hidden behaviors is important for understanding the malware's malicious behavior and for effective malware defense.

MineSweeper [7] is a tool that automatically explores multiple execution paths, in order to uncover the hidden functionality in malware. This tool takes advantage of the symbolic execution functionality in TEMU. It marks potential trigger conditions (e.g., inputs from the registry, file system, network, and system time) as symbolic, and then performs symbolic execution at runtime. Whenever a branch condition becomes symbolic, this tool determines feasible branches by solving the path predicate for each branch, and then explores these feasible branches.

Program	Run Time		Cond.	Symbolic
	Total	STP	Jumps	Ratio
MyDoom	28m	2.2m	11	0.00136%
NetSky	9m	0.3m	6	0.00040%
Perf. Keylogger	2m	<0.1m	2	0.00508%
TFN	21m	6.5m	14	0.00052%

Table 4: Analysis Results on Real-world Malware Samples using MineSweeper

Experimental results. Four realworld malware samples, including two Email worms (MyDoom and NetSky), a keylogger (Perfect Keylogger) and a DDoS tool (TFN), were used to evaluate the effectiveness and efficiency of MineSweeper. MineSweeper was able to uncover hidden functionalities in these malware samples. Table 4 summarizes the results. The end-to-end run time was up to 28 minutes. In addition, the run time for solving formulas (i.e., STP runtime) was listed, up to 2.2 minutes. The number of conditional jumps that need to be explored symbolically was relatively small, up to 14. The percentage of instructions that need to be executed symbolically was also low.

The NetSky worm was known to have time triggered functionality. MineSweeper extracted from NetSky a graph of program paths that depend on the system time, shown in Figure 6. We can see that the date must be February 26th, 2004, between 6 and 9am, to launch an attack.

Therefore, by making use of symbolic execution functionality in TEMU, MineSweeper can automatically extract hidden functionality in malware in minutes.

6.5 Tool-writing Ease

	LOC
TEMU	315,174
Renovo	364
Panorama	793
HookFinder	1,052
MineSweeper	3,652

Table 5: Code Sizes of TEMU and Plugins

We use code sizes to roughly measure the amount of efforts that are needed to build these analysis tools. Table 5 lists the lines-of-code for each of these analysis tools. We can see that with several hundred or thousand lines of code, various comprehensive analysis tools can be easily built on top of our analysis platform. As a comparison, we also show the size of the analysis platform. TEMU is based on QEMU 0.9.1. TEMU has an addition of 11,097 lines to the QEMU code base, which has 304,077 lines of code in total. Therefore, TEMU provides common functionalities and hides the unnecessary complexity for various analysis purposes. Compared to writing a tool from scratch, the benefit of this analysis platform is clear.

7. RELATED WORK

Tools like DynamoRIO [5], Pin [25], Strata [36], and Valgrind [29] support fine-grained instrumentation of a user-level program. They all provide high-level interfaces to facilitate building custom analysis tools. However, as they can only instrument a single user-level process, they are not suitable to analyze the activities in the operating system kernel (e.g., kernel malware and kernel vulnerabilities) or applications that involve multiple processes. DynamoRIO, PIN and Strata are designed to achieve highly efficient dynamic instrumentation, whereas Valgrind is targeted at heavyweight analysis, such as shadow value analysis [28]. It eliminates the complexity of the x86 instruction set by first translating x86 instructions into intermediate representations, namely VEX statements. Like Valgrind, TEMU is designed to support myriad of in-depth analyses. It takes advantage of dynamic translation mechanism in QEMU to build more comprehensive analysis platform.

PinOS [10] is an extension to Pin for whole-system instrumentation. Hence, PinOS can be used to instrument both kernel and user-level code. To achieve whole-system instrumentation, PinOS is built on top of the Xen [1] virtual machine monitor with Intel VT technology. TEMU also enables whole-system instrumentation and offers a whole-system view. However, compared to PinOS, TEMU provides substantially better support, such as OS-level semantics extraction and layered annotative execution, for analysis tools built on top of it.

Dytan [14] is a platform for performing dynamic taint analysis. With the high-level interface provided by Dytan, custom taint analysis tools can be easily built. Dytan is based on Pin, and thus can only analyze a single user-level process. By contrast, TEMU is a whole-system analysis platform and provides even richer functionalities for various analysis purposes.

Nirvana [3] is another analysis platform for a user-level program, using software dynamic translation techniques. In particular, it is used to build a sophisticated instruction-level tracing and time-traveling debugging tool, called iDNA. A

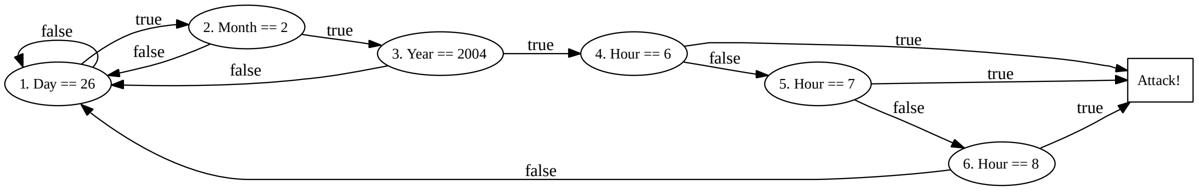


Figure 6: NetSky's Hidden Behavior Extracted by MineSweeper.

comprehensive tracing tool was also developed on top of TEMU. While Nirvana can only analyze a user-level program, TEMU can be used to analyze kernel code and multiple programs simultaneously, with extra functionalities, such as exploit detection, offline symbolic execution, etc.

Cobra [39] is a malware analysis platform. Cobra is implemented as a kernel module and inserted into the Windows kernel space to observe malware's execution in both user and kernel space. It uses a technique called localized execution to instrument and inspect malware's behavior. The localized execution technique is in spirit similar to dynamic translation techniques. Cobra takes an internal approach, because the analysis is performed in the same execution environment to be analyzed.

Ether [18] is another platform for malware analysis. Ether makes use of hardware virtualization techniques to observe malware's execution in a stealthy manner. Compared with Cobra, Ether takes an external approach. The analysis is performed outside the guest system, which in principle has better transparency than an internal approach. However, Ether is not an ideal platform for in-depth malware analysis, which requires instruction-level instrumentation. Although fine-grained instrumentation can be achieved through single-step mode, its significant performance overhead (hundreds of times slowdown) is unacceptable in many cases. In contrast, by using dynamic translation techniques, TEMU can perform in-depth malware with much better efficiency due to dynamic binary translation. However, emulation via dynamic translation is not as transparent as hardware virtualization [19, 26, 34]. We leave it as future work to build a more stealthy and efficient analysis platform by combining hardware virtualization and emulation techniques.

8. CONCLUSION AND FUTURE WORK

In this paper, we presented an extensible analysis platform, TEMU. In order to support a wide-spectrum of dynamic binary analysis needs, TEMU explores a unique design space. It is able to provide a whole-system view, perform out-of-the-box analysis, and bridge the semantic gap between hardware-level and OS-level views. To further facilitate fine-grained analysis, TEMU incorporates shadow flag analysis, taint analysis, and symbolic execution, and generalizes these techniques into a unified technique, layered annotative execution. To demonstrate the capabilities of this platform, we described four malware analysis tools built on top of it. These tools were comprehensive and powerful by utilizing different functionalities of TEMU. Moreover, the efforts of implementing these tools were significantly alleviated, due to the support of TEMU.

We plan to enhance TEMU in the following aspects: 1) better transparency by combining hardware virtualization and dynamic translation techniques; 2) better efficiency by exploiting more sophisticated optimization techniques; and 3) more robust OS-level semantic view reconstruction.

9. REFERENCES

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03*, pages 164–177, 2003.

[2] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX 2005 Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[3] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE'06)*, pages 154–163, 2006.

[4] BitBlaze: Binary analysis for computer security. `http://bitblaze.cs.berkeley.edu/`.

[5] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization (CGO'03)*, March 2003.

[6] David Brumley, Cody Hartwig, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Dawn Song. BitScope: Automatically dissecting malicious binaries. Technical Report CS-07-133, School of Computer Science, Carnegie Mellon University, March 2007.

[7] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Pongsin Poosankam, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In *Book chapter in "Botnet Analysis and Defense", Editors Wenke Lee et. al.*, 2007.

[8] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 2–16, 2006.

[9] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 29th IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.

[10] Prashanth P. Bungale and Chi-Keung Luk. PinOS: A programmable framework for whole-system dynamic instrumentation. In *Proceedings of the 3rd international conference on Virtual Execution Environments (VEE'07)*, pages 137–147, 2007.

[11] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM Conference on Computer and Communication Security*, Chicago, IL, November 2009.

[12] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of ACM Conference on Computer and Communication Security*, October 2007.

[13] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th USENIX Security Symposium (Security'03)*, August 2004.

[14] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis (ISSTA'07)*, pages 196–206, 2007.

[15] Jedidiah Crandall, Zhendong Su, S. Felix Wu, and Frederic Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proc. 12th ACM Conference on Computer and Communications Security (CCS)*, 2005.

[16] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO'04)*, December 2004.

[17] Data Rescue. Universal PE Unpacker plug-in. `http://www.datarescue.com/idabase/unpack_pe`.

[18] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 51–62, 2008.

[19] Peter Ferrie. Attacks on virtual machine emulators. Symantec Security Response, December 2006.

[20] Vijay Ganesh. STP: A decision procedure for bitvectors and arrays. `http://theory.stanford.edu/~vganesh/stp.html`, 2007.

[21] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In W. Damm and H. Hermanns, editors, *Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 524–536, Berlin, Germany, July 2007. Springer-Verlag.

[22] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of Network and Distributed Systems Security Symposium (NDSS'03)*, February 2003.

[23] Google's desktop search red flag. `http://www.internetnews.com/xSP/article.php/3584131`.

[24] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 5th ACM Workshop on Recurring Malcode (WORM)*, October 2007.

[25] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of 2005 Programming Language Design and Implementation (PLDI) conference*, june 2005.

[26] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing cpu emulators. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA'09)*, pages 261–272, 2009.

[27] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy(Oakland'07)*, May 2007.

[28] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual Execution Environments (VEE '07)*, pages 65–74, 2007.

[29] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100, 2007.

[30] James Newsome, David Brumley, Jason Franklin, and Dawn Song. Replayer: Automatic protocol replay by binary analysis. In Rebecca Write, Sabrina De Capitani di Vimercati, and Vitaly Shmatikov, editors, *In the Proceedings of the* 13^{th} *ACM Conference on Computer and and Communications Security (CCS)*, pages 311–321, 2006.

[31] James Newsome, David Brumley, Dawn Song, Jad Chamcham, and Xeno Kovah. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *Proceedings of the* 13^{th} *Annual Network and Distributed System Security Symposium (NDSS)*, 2006.

[32] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, February 2005.

[33] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks. In *EuroSys 2006*, April 2006.

[34] Thomas Raffetseder, Christopher Krügel, and Engin Kirda. Detecting system emulators. In *Information Security, 10th International Conference, ISC 2007*, pages 1–18, October 2007.

[35] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 289–300, Washington, DC, USA, 2006. IEEE Computer Society.

[36] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the international symposium on Code generation and optimization (CGO'03)*, pages 36–47, Washington, DC, USA, 2003.

[37] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, December 2008.

[38] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*, October 2004.

[39] Amit Vasudevan and Ramesh Yerraballi. Cobra: Fine-grained malware analysis using stealth localized-executions. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 264–279, Washington, DC, USA, 2006. IEEE Computer Society.

[40] Xuxian Jiang Xinyuan Wang and Dongyan Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *Proceedings of ACM Conference on Computer and Communication Security*, October 2007.

[41] Heng Yin, Zhenkai Liang, and Dawn Song. HookFinder: Identifying and understanding malware hooking behavior. In *15th Annual Network and Distributed System Security Symposium*, 2008.

[42] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of ACM Conference on Computer and Communication Security*, October 2007.

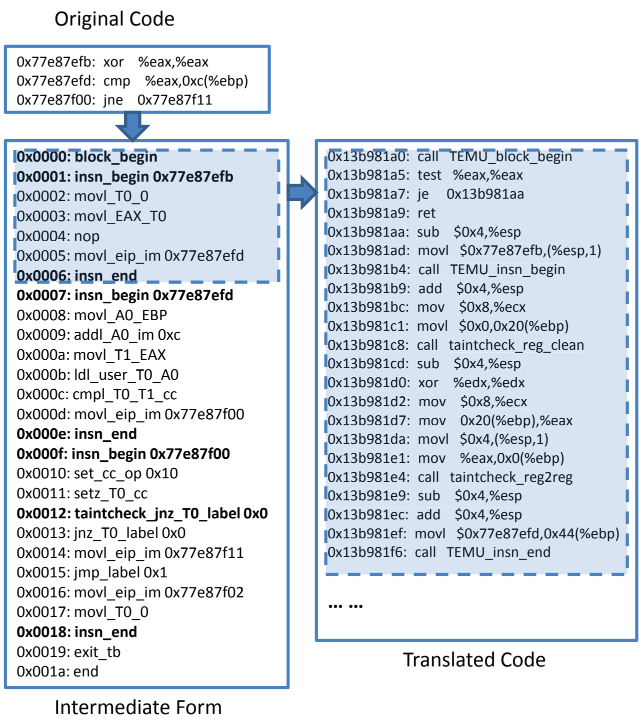
APPENDIX

A. CODE INSTRUMENTATION IN TEMU

Here we walk through the code instrumentation process in TEMU, by using a concrete example shown in Figure 7. In this example, the original code block has three instructions, ending with a conditional jump. This original code block is first translated into a list of statements in an intermediate form. One instruction may be translated into multiple intermediate statements. For instance, the shaded area in the central text block corresponds to the statements for the first instruction. To instrument this code block, we have two approaches. First we extend the implementation of existing statements.

```
void OPPROTO op_movl_T0_0(void)
{
  T0 = 0;
  taintcheck_reg_clean(R_T0);
}
```

The above code snippet shows the implementation of the statement `movl_T0_0`. In addition to assigning 0 to the temporary register T0, we insert a function call `taintcheck_reg_clean` to clean up the annotation that may be associated with T0. The second approach is to define and insert new statements. The statements in bold font are new statements inserted during instrumentation. For instance, `block_begin` is inserted for the start of a basic block, and `insn_begin` and `insn_end` are inserted at instruction boundary. Eventually, the intermediate statements will be translated into a block of instructions to be executed on the host machine. For brevity, Figure 7 only shows the translated host instructions that correspond to the first guest instruction. The inserted statements are implemented as function calls, such as `TEMU_block_begin` and `TEMU_insn_begin`.



```
0x77e87efb:  xor   %eax,%eax
0x77e87efd:  cmp   %eax,0xc(%ebp)
0x77e87f00:  jne   0x77e87f11
```

```
0x0000: block_begin
0x0001: insn_begin 0x77e87efb
0x0002: movl_T0_0
0x0003: movl_EAX_T0
0x0004: nop
0x0005: movl_eip_im 0x77e87efd
0x0006: insn_end
0x0007: insn_begin 0x77e87efd
0x0008: movl_A0_EBP
0x0009: addl_A0_im 0xc
0x000a: movl_T1_EAX
0x000b: ldl_user_T0_A0
0x000c: cmpl_T0_T1_cc
0x000d: movl_eip_im 0x77e87f00
0x000e: insn_end
0x000f: insn_begin 0x77e87f00
0x0010: set_cc_op 0x10
0x0011: setz_T0_cc
0x0012: taintcheck_jnz_T0_label 0x0
0x0013: jnz_T0_label 0x0
0x0014: movl_eip_im 0x77e87f11
0x0015: jmp_label 0x1
0x0016: movl_eip_im 0x77e87f02
0x0017: movl_T0_0
0x0018: insn_end
0x0019: exit_tb
0x001a: end
```

```
0x13b981a0:  call   TEMU_block_begin
0x13b981a5:  test   %eax,%eax
0x13b981a7:  je     0x13b981aa
0x13b981a9:  ret
0x13b981aa:  sub    $0x4,%esp
0x13b981ad:  movl   $0x77e87efb,(%esp,1)
0x13b981b4:  call   TEMU_insn_begin
0x13b981b9:  add    $0x4,%esp
0x13b981bc:  mov    $0x8,%ecx
0x13b981c1:  movl   $0x0,0x20(%ebp)
0x13b981c8:  call   taintcheck_reg_clean
0x13b981cd:  sub    $0x4,%esp
0x13b981d0:  xor    %edx,%edx
0x13b981d2:  mov    $0x8,%ecx
0x13b981d7:  mov    0x20(%ebp),%eax
0x13b981da:  movl   $0x4,(%esp,1)
0x13b981e1:  mov    %eax,0x0(%ebp)
0x13b981e4:  call   taintcheck_reg2reg
0x13b981e9:  sub    $0x4,%esp
0x13b981ec:  add    $0x4,%esp
0x13b981ef:  movl   $0x77e87efd,0x44(%ebp)
0x13b981f6:  call   TEMU_insn_end

... ...
```

Figure 7: An example of how TEMU instruments code.