Operating System Doppelgängers: The Creation of Indistinguishable, Deterministic Environments

Jason L. Wright
Thought Networks, LLC

Chris Eagle
Naval Postgraduate School

Tim Vidas
0x90 Labs

Abstract

In support of the Cyber Grand Challenge program, DARPA required an execution environment that not only focused research, but also facilitated rigorous, repeatable measurements and presented the highest levels of integrity. The desire was that measurements published out of the program could provide a basis for years of research in automated cyber reasoning. Further, due to the competitive nature of the program, and the preexisting cultural acceptance of undermining the integrity of cybersecurity competitions, there was a real need to create a system presenting very limited risk surface. The DARPA Experimental Cyber Research Evaluation Environment, or DECREE, was conceived to meet these needs.

DECREE is a novel environment specification consisting of a simplified set of system calls and a consistent, measurable interface to system resources (CPU, memory, etc) for every user program. In this work we explore many of the decisions and constraints that drove the design and development of DECREE, and, specifically, detail implementations of DECREE that have already been deployed in high-profile, public experiments. In particular, we detail the precise measures taken to ensure that two disparate implementations of the DECREE specification (Linux and FreeBSD) are indistinguishable from one another to user programs, while simultaneously striving for the highest levels of determinism and fidelity for measurement. In the rare situations where the operating system does not provide enough capability, we detail the use of a custom hypervisor to address the deficiencies.

1 Introduction

The US Department of Defense (DoD) Defense Advanced Research Projects Agency (DARPA) created the Cyber Grand Challenge (CGC) to push the edge of technology in "autonomous cyber defense capabilities that combine the speed and scale of automation with reasoning ability exceeding those of human experts [5]." The challenge was framed around software vulnerabilities in binary software. To encourage focused research, a concentrated software environment was a crucial tenet of the challenge. At the same time, due to the competitive nature of the challenge, many architecture and implementation design decisions made throughout program aimed to ensure the highest standards of integrity. For these reasons, there was a need for a simplified and consistent environment for executing challenge binaries and thus the DECREE (the DARPA Experimental Cyber Research Evaluation Environment) specification was devised.

An aspect of the CGC vision realized in DECREE is the creation of an environment where an alternative, minimal software ecosystem could be created such that it mirrored the challenges and constraints of real-world defenders. One way DECREE implements such an environment is a custom system call[1] interface. Modern operating systems implement hundreds of system calls (e.g. Linux now employs nearly 400). Correctly modeling all system calls is an arduous task for program analysis. DECREE specifies a reduced set of just seven system calls, which provide enough expressiveness to represent real-world programs and flaws while simultaneously simplifying the modeling problem for CGC competitors.

When executed, each DECREE binary is presented with a consistent view of resources (e.g. CPU, operating system services, memory) in a way that is measurable and repeatable. Further, the lone source of randomness available to DECREE programs is actually pseudorandom and fully under the control of the kernel. When seeded identically, two instantiations of the same program should yield identical results when presented with identical inputs. The primary goal for specifying DECREE in this manner is to provide a platform on which instrumented execution of programs can provide repro-

[1] A system call represents a request on behalf of a user program for a service provided by the kernel.

Table 1: DECREE system calls

Number	System Call	UNIX™ analog
1	_terminate	_exit
2	transmit	write
3	receive	read
4	fdwait	select
5	allocate	mmap
6	deallocate	munmap
7	random	

ducible measurements of program performance. Consequently, when a DECREE program is executed and its system calls are recorded, all required state is available to reliably produce and verify the same results on similar (but not necessarily identical) hardware.

This paper details a novel, comprehensive method to provide a consistent view of CPU, memory, and operating system interfaces to allow repeatable measurement of program execution to a degree never provided by any open source operating system. The design and methods are applicable to future environments that require a high standard for consistency. We identify instructions and circumstances that violate two core properties required of our specification: *determinism* and *indistinguishability*. Further, we describe the process by which two disparate operating systems (one under 32-bit Linux and the other under 64-bit FreeBSD) are adjusted to satisfy both properties; each pedantically diverging from their origin and converging upon the DECREE specification.

The rest of the paper is organized as follows. Section 2 describes DECREE, loading DECREE binaries, and its system calls. Section 3 describes our DECREE implementation details with emphasis on the different approaches implemented between the CGC Qualifying Event (CQE) and the CGC Final Event (CFE). Section 4 provides discussion and relationship to previous work. Section 5 provides the conclusion.

2 Overview of DECREE

DECREE was conceived as an experimental operating system interface expressive enough to present real world program analysis challenges yet simple enough to model so that automated program analysis tools could enumerate the underlying system more easily than they could a "full" operating system. As a result, DECREE has just seven system calls, as enumerated in Table 1. DECREE binaries are distinguished from other, "native" OS binaries, and conform to a simplified file format based on ELF (Executable and Linking Format, [1]).

Each DECREE binary, when loaded, is presented with a consistent view of the CPU, operating system services, and memory in a way that is measurable and repeatable. The CPU chosen for modeling is the 4th generation Intel Core micro-architecture (i.e. "Haswell") running in 32-bit mode (a few specific instructions are not modeled as described in Section 3.2). Operating system services are requested via software interrupt (`INT 0x80`). Virtual address space isallocated to demand page the binary and all pages allocated during loading are readable and optionally writeable and/or executable. Pages are automatically allocated for the stack and are executable. Heap memory can be allocated and deallocated, and memory allocated this way is always readable and writeable and optionally executable. Lastly, all sources of entropy other than DECREE's `random()` system call have been made unavailable to DECREE executables.

No mechanism exists to create a new process or thread (`fork()`) or to execute new programs (`execve()`). Further, there is no way to create new files or sockets. Instead, DECREE binaries require file descriptors to be configured by native processes which then fork and execute DECREE processes that inherit a pre-configured set of I/O descriptors. As such, DECREE is implemented as a special binary loader based on the ELF loader already present in the native operating system. This loader is responsible for reading information about the layout of the program from its corresponding file and ensuring that text and data pages of the process are mapped to the specified memory addresses. DECREE binaries are statically linked; shared objects and dynamic loading are not supported.

The calls `_terminate()`, `transmit()` and `receive()` correspond directly to the native system calls `_exit()`, `write()` and `read()`, respectively. The `fdwait()` function is analogous to the `select()` system call which allows programs to wait until data is ready to be read from or written to a file descriptor or until a certain amount of time has passed (the ability to wait for exceptional conditions is removed since such exceptional conditions do not exist within DECREE).

The `allocate()` system call provides a subset of the functionality offered by the standard `mmap()` call. Unlike `mmap()`, only anonymous pages can be allocated, all pages are readable and writeable (the caller can specify whether the page is also executable), and the caller cannot specify a preferred virtual address for the mapping. Pages allocated with `allocate()` are simply a reservation of virtual address space; backing physical pages are not allocated until the virtual addresses are accessed by the process. DECREE offers no capability to modify an existing mapping such as that offered by `mprotect()`.

The `deallocate()` call is analogous to `munmap()`; pages allocated during the loading of the binary and pages allocated with `allocate()` can all be deallocated.

The lone exception to this is a single page of read-only, random data mapped into every DECREE process at process creation time. This page is used to evaluate proofs of vulnerability and cannot be deallocated. If a backing page is allocated for a virtual mapping, the `deallocate()` call returns the page to the system. The freed virtual address space will cause a fatal fault if accessed after having been deallocated.

3 Implementation of DECREE

DECREE is implemented on top of and in terms of Linux and FreeBSD interfaces. This section examines the environment exposed to DECREE programs and how that environment is made consistent across implementations.

3.1 Loading DECREE binaries

The DECREE binary format is heavily based on that of ELF, but without support for shared objects or dynamic interpreters. The DECREE loader for both Linux and FreeBSD was based on their respective loaders for ELF binaries. One major difference between DECREE processes and native processes is the lack of environment variables and command line arguments normally made available to each newly created process.

To facilitate detection of memory disclosure vulnerabilities, a page filled with the output of a Pseudorandom Number Generator (PRNG) is mapped into each process. If a proof of vulnerability provides knowledge of 4 consecutive bytes from this page, it is considered to have disclosed privileged information from the binary. When a DECREE binary is executed, a PRNG seed value may be specified using `execve()` arguments. Once the loader has mapped all other pages into the new process image, the memory disclosure page is mapped at the fixed address of `0x4347c000` and its contents filled from the seeded PRNG. To facilitate moving this page to other locations, the initial value of the ECX general purpose register for each DECREE process is set to the virtual address of this page. This page of memory is marked read-only and may not be unmapped by `deallocate()`.

3.2 Determinism and indistinguishability

Certain instructions available to user processes (i.e. unprivileged) may provide information about the specific processor on which a binary is executing (e.g. CPUID, Section 3.2.1), internal processor state (e.g. RDTSC, Section 3.2.2), and the operating environment in which the binary is running (e.g. LAR, Section 3.2.3 and SGDT, Section 3.2.4). Such instructions must behave consistently across all DECREE implementations. Moreover, any implementation of the DECREE specification must exhibit *determinism*, repeatable execution and the absence of any source of external entropy, and *indistinguishability*, the inability to discern one implementation of the DECREE specification from another. The remainder of this section describes implementation details to satisfy these two properties.

3.2.1 CPUID

The CPUID instruction was added to IA-32 with the i486SL and Pentium generation of processors. It allows an unprivileged process to query information about the processor. Values available via CPUID can identify the APIC ID (which processor in a multiprocessor system), stepping, clock rate, supported features, etc., potentially allowing a user process to distinguish between two different systems. Our implementation ensures constant return values from CPUID using modified or custom hypervisors.

The values returned by our implementation mimic the values of a uniprocessor VirtualBox 4 Virtual Machine (VM) running our Linux distribution on a MacOS host. These values were chosen for expediency. Some small modifications to the indications of specific features were changed and those changes are documented in the following sections where appropriate.

For Linux running under the Xen hypervisor, code is added to the existing handler for CPUID related VMExits (CPUID is an instruction that unconditionally causes an exit to the hypervisor). For FreeBSD and our custom hypervisor, it is possible to detect whether the CPUID exit is caused by a DECREE process, the kernel, or a native process. DECREE processes receive CGC normalized values, while the kernel and native processes receive the actual results from calling CPUID.

3.2.2 RDPMC/RDRAND/RDSEED/RDTSC/RDTSCP

Several potentially unprivileged instructions which return state information available only to the current CPU or current CPU thread are enumerated in Table 2. Each of these instructions could be used as an external source of entropy, violating the determinism property. Accordingly, they are each disabled in our implementation as described below.

The problem with these instructions from the point of view of DECREE is the lack of reproducibility of results. Given an assembly sequence like that shown in Figure 1, where does execution continue: `label1` or `label2`? It depends strictly on state visible only by the CPU executing the process when it is executing: no time before or after. Each branch will be taken approximately 50% of the time, but there is no way to predict a priori or post

```
RDSEED          # EDX:EAX <- PRNG value
TEST 1, %eax    # Check bit 0 of EAX
JZ label1       # Jump if EAX.0 is clear
JMP label2      # Jump if EAX.0 is set
```

Figure 1: Is the first branch taken or not?

mortem whether the branch will be or was taken.

One possible solution to the problem is to intercept the instructions and return constants or known streams of values. It was decided that disabling these instructions met our goals: we do not want to provide external entropy or timing information to DECREE processes, so returning a fault to a process requesting access to a source of entroy, the performance counters or the time stamp counter fits with that model.

RDPMC, RDTSC, and RDTSCP can all be disabled in user processes by setting or clearing bits in configuration register 4 (CR4). If set, CR4.TSD disables both RDTSC and RDTSCP in non-kernel processes. If clear, CR4.PCE disables RDPMC in non-kernel processes. For both Linux and FreeBSD, the context switch code was modified to ensure these instructions were disabled for DECREE processes. On recent Intel processors, RDTSCP has its own VM Exit enable bit. If this bit is not set, and CR4.TSD is set, RDTSCP generates an illegal instruction while RDTSC generates a general protection fault. In order for both instructions to generate general protection fault, the RDTSCP VM Exit must be enabled and CR4.TSD must be set.

RDRAND and RDSEED cannot be disabled via processor configuration registers, but both instructions can generate VM Exits and thus be handled by a hypervisor. In our modified Xen hypervisor and our custom hypervisor, exiting based on RDRAND and RDSEED is enabled. In the hypervisor, if the source is found to be a DECREE process, an illegal instruction is generated in the guest operating system. Otherwise, the native instruction is executed and the results passed back to the guest kernel or process. Both hypervisors also remove indications from CPUID that these instructions are supported by ensuring the CPUID.01H:ECX.RDRAND and CPUID.(EAX=07H,ECX=0H):RBX.RDSEED bits are clear.

Table 2: Instructions Capable of Acting as an External Source of Entropy

Instruction	Description
RDPMC	Read performance monitoring counter
RDRAND	Read random number generator
RDSEED	Read random number generator
RDTSC	Read time-stamp counter
RDTSCP	Read time-stamp counter and processor ID

Table 3: Using LSL, LAR, VERR, and VERW to probe segments

Seg.	LAR	LSL	R/W	Operating System
0	0xf8ec00	0x4	—	FreeBSD 10.2 i386
3	0xcffb00	0xffffffff	r	
4	0xf8ec00	0x24	—	
5	0xcff300	0xffffffff	r/w	
16	0xf8ec00	0x84	—	
6	0xdff300	0xffffffff	r/w	Linux 3.13.2 i386
14	0xcffb00	0xffffffff	r	
15	0xcff300	0xffffffff	r/w	
4	0xcffb00	0xffffffff	r	Windows 8.1 32bit
5	0xcff300	0xffffffff	r/w	
6	0x40f300	0xfff	r/w	
10	0xf200	0xffff	r/w	
4	0xcffb00	0xffffffff	r	Windows 7 64bit
5	0xcff300	0xffffffff	r/w	
6	0x20fb00	0x0	r	
10	0x40f300	0x3c00	r/w	
14	0xcffb00	0xffffffff	r	DECREE
15	0xcff300	0xffffffff	r/w	

3.2.3 Segments

Addresses in Intel 64 and IA-32 are implicitly or explicitly formed from a segment and an offset. An instruction such as `MOV (%eax),%ebx` which moves the value stored at the address stored in EAX into the register EBX implicitly uses the data segment register, DS. Stack instructions implicitly use the stack segment (SS) and references to executable code implicitly use the code segment (CS).

Each segment has associated with it a set of access rights (writeable, executable), base address, and limit (length). Several instructions allow for probing the configuration of segments by user mode processes:

- LAR — Load Access Rights,
- LSL — Load Segment Limit, and
- VERR/VERW — Verify segment for read / write.

Operating systems may be profiled (violating the indistinguishability property) by probing for the existence of segments and examining the configuration of each. Table 3 shows the values returned by LSL, LAR, VERR, and VERW by several operating systems/configurations. The visible segments are sufficient to differentiate FreeBSD (0, 3, 4, 5, 16), Linux (6, 14, 15), and Windows (4, 5, 6, 10), but it is necessary to examine the segment configuration to differentiate 32bit and 64bit Windows.

DECREE exposes a miminal set of descriptor table entries to the user process: a code segment (14, $CS = 0x73$) and a data segment (15, $DS = ES = FS = GS = SS = 0x7b$). Under Linux, this is implemented by the CGC

loader by forcing FS/GS to contain the value __USER_DS during loading and by ensuring that thread local storage is initialized to zero (unused). Marking the thread local storage as unused has the side effect of marking segment 6 as not present.

In the 64-bit FreeBSD implementation, more drastic changes are required. The first step is to rearrange the order of the segments to match those of Linux. In so doing, the LDT is marked as not present; nothing used in CGC requires the LDT, so this is acceptable. Further, code was added to the context switch to toggle the visibility of the 64bit code segment and the 32bit FS/GS segments.

3.2.4 Descriptor Tables

Four registers contain references to descriptor tables. The Global Descriptor Table Register (GDTR) contains the linear (physical) address of the Global Descriptor Table and the length of the table. The GDT, in turn, has entries for each segment and "special descriptor" such as the Local Descriptor Table (LDT), Task Descriptor (TD), etc. The Interrupt Descriptor Table Register (IDTR) contains the linear address and length of the Interrupt Descriptor Table (IDT). The Task Register (TR) and Local Descriptor Table Register contain entry numbers within the GDT for the Task Descriptor and Local Descriptor Table entries, respectively.

A user process is not able to change the value of these registers, but four instructions allow for querying their current values:

- SGDT — Store GDTR,
- SIDT — Store IDTR,
- SLDT — Store LDTR, and
- STR — Store TR.

Some of these registers have been used as virtual machine detection methods: SIDT [11; 21], SLDT [18], and SGDT [12]. As a result, Linux attempts to reduce the disclosure by using a constant value for IDTR (reference Linux source). However, the GDTR and TR are left exposed (LDT is zero, unused).

It is sufficient in some cases to differentiate host operating system and virtual machine monitor (either would violate the property of indistinguishability) presence using SGDT, SIDT, SLDT, and STR. Values for various operating systems / configurations are shown in Table 4. In each case the virtual environment is configured for a single processor. Linux, for example, uses a separate GDT for each processor, so the value returned by SGDT varies with the processor on which the instruction is executed.

With normal kernel level controls, it is not possible to prevent a user process from reading and interpreting the GDTR, IDTR, LDTR, and TR. However, using a hypervisor, VMExits (transitions from guest operating system to hypervisor) can be generated when instructions that load or store these registers are executed inside the guest. The hypervisor then has the option to generate a trap (e.g. illegal instruction trap) or process the instruction. While we could have simply killed user processes executing the descriptor table instructions, our implementation of DECREE allows them, but always returns a constant value. Specifically our implementation returns the same value observed by running our VM in VirtualBox (the documented configuration).

For Linux (used in the CGC qualifying event), we modified the Xen hypervisor to generate VMExits for descriptor table manipulation instructions and wrote the corresponding code for handling the exits. This code decodes the instructions and detects whether the guest kernel, guest native process, or guest DECREE process is executing the instruction. For guest kernel and guest native, the actual values for the descriptor table registers are returned. For DECREE processes, the constants from Table 4 are used. Since the user process cannot verify the values, it cannot detect the deception.

For FreeBSD (used in the CFE), we used a custom written "blue-pill" style hypervisor [22]. VMExits were configured for the descriptor table instructions. The hypervisor was transparent to the guest (blue-pilled) kernel and native guest processes, but provided our deceptive values to any DECREE processes in order to paint the desired, consistent, environmental picture.

3.2.5 SMSW

The SMSW (Store Machine Status Word) instruction is provided for backwards compatibility with the Intel 286 processor. It is an unprivileged instruction which allows the examination of configuration register 0 (CR0). Modern software is supposed to use the `MOV CRn` family of instructions to read and/or write CR0 which are privileged instructions. Table 5 shows the results of using SMSW on various operating systems; it is possible to differentiate some operating systems and configurations using the MSW value as discussed in [20; 19]. Consequently, our implementation must address SMSW to satisfy the indistinguishability property.

Hypervisors can catch read and write accesses to the configuration registers and handle them. Since the value DECREE needed to present to user processes matched the value already returned, no special handling is required. To make this invariant, our hypervisor ensures that all reserved or undefined bits are forced to constant values using guest/host masks and read shadows provided by Intel VT-x.

Table 4: Using GDTR, IDTR, LDTR, and TR for identification

GDTR		IDTR				
base	**limit**	**base**	**limit**	**LDTR**	**TR**	**Operating System**
0xf355c000	0xffff	0xd0ae3d60	0x07ff	0x18	0x1f8	OpenBSD 5.9 i386
0xc142cfa4	0x0097	0xc13e2aa5	0x07ff	0x48	0x050	FreeBSD 10.2 i386
0x041bc000	0x007f	0x041bc080	0x0fff	0x00	0x040	Windows 7 64bit
0x8135f000	0x03ff	0x8135f400	0x07ff	0x00	0x028	Windows 8.1 32bit
0xfb7fe000	0x01ef	0xfb7fd000	0x0fff	0x00	0x070	OpenIndiana
0xdf24a000	0x00ff	0xc13de000	0x07ff	0x00	0x080	Linux 3.13.2 32bit (Xen, HVM)
0x00000000	0xefff	0x802f4340	0xffff	0x00	0x080	Linux 3.13.2 32bit (Xen, domU)
0xf7beb000	0x00ff	0xfffba000	0x07ff	0x00	0x080	Linux 3.13.2 32bit (VirtualBox)
0xf7beb000	0x00ff	0xfffba000	0x07ff	0x00	0x080	DECREE

Table 5: Examples of Machine Status Words

MSW	Operating System
0x80010031	Windows XP SP3 i386 (VM) Windows Vista i386 (VM)
0x80010033	Windows 8.1 SP1 i386 (VM) MacOS X (Mavericks)
0x8001003b	OpenBSD/i386 5.9 FreeBSD/i386 10.2 DECREE Linux/i386 3.13.2
0x8005003b	OpenIndiana (VM)
0x8005f031	Windows 7 AMD64

3.2.6 Floating point, MMX, SSE, etc.

All modern Intel processors support floating point instructions, MMX, SSE, SSE2, etc. We needed to verify that floating point operations would be performed as consistently as possible across our processors and base operating systems. Fortunately, the processors were of the same micro-architecture. In unit testing, it was observed that FreeBSD did not fully zero the initial state of the XMM registers when creating a new FPU state. This was fixed and a patch submitted and accepted by the FreeBSD team. Also, for FreeBSD/i386, the floating point control word was configured for Double Precision (53 bits) instead of Double Extended Precision (64 bits) and the infinity control bit was set (for compatibility with the Intel 287). Our unit tests verified the configuration of Double Extended Precision and infinity control bits (the default on FreeBSD/AMD64, Linux/i386, and Linux/AMD64).

3.2.7 SYSENTER

In Linux/i386, two methods are supported for executing system calls: the older "INT $0x80" method, and SYSENTER/SYSEXIT method introduced with the Intel Pentium II. The "INT $0x80" method requires little cooperation from a user process: the system call arguments are saved into specific registers and the interrupt instruction is used. The interrupt automatically saves EIP, CS, and EFLAGS and transitions to the kernel. To return from kernel, IRET loads the saved EIP, CS, and EFLAGS and transitions back to the user.

SYSENTER/SYSEXIT saves nothing automatically. User processes are required to save any state required and restore it upon return. Linux supports this by mapping system enter/exit code for user processes into a page shared by all user processes. The enter code pushes ECX, EDX, EBP and calls SYSENTER. When the system call is complete, the corresponding SYSEXIT "returns" to this mapped page to pop EBP, EDX, ECX and return to the calling process.

For our implementation, both methods could have been supported, but it was decided for simplicity's sake to only support the interrupt method. To support SYSENTER / SYSEXIT without a constant return address would have required copying data from the stack in the user process or other contortions. Instead, the Linux kernel is modified such that when SYSENTER is executed by a DECREE process, the process is killed with an illegal instruction signal.

FreeBSD, on the other hand, has no support for SYSENTER / SYSEXIT. As a result, processes using this instruction are still killed, but with the wrong signal: SIGSEGV. This happens because if SYSENTER MSRs are not configured, execution of the instruction generates a general protection fault. To ensure delivery of the correct signal, enough support for SYSENTER is added to the FreeBSD kernel to catch the entry to the kernel from SYSENTER and kill the offending process with an illegal instruction signal.

3.2.8 The `fdwait()` system call

The Linux implementation of `select()` which is used to implement `fdwait()` normally modifies the provided timeval structure to indicate the amount of time remaining if the `select()` call returns before the timeout expires. Returning this modified value would violate the determinism property because the timing information is an external source of entropy. Fortunately Linux does

not modify this value if the "sticky select" bit is set in the processes personality field. The DECREE personality value is defined with this bit set in order to obtain this behavior on Linux. This is the default behavior on FreeBSD; no modifications were required.

3.3 Linux Implementation

DECREE processes do not support signal handling. Instead, the Linux signal code is modified to cause any signal sent to a DECREE process to cause the process to terminate. One signal, SIGPIPE, generated when a process attempts to write to a closed file descriptor is excepted (the process can check for EPIPE error returns instead).

Normally, when Linux is out of memory (a request for a backing page has come in, but no pages are available on the free list), it will rank the current processes and kill one of the larger processes. In our threat model, all DECREE processes operate at the lowest level of trust. To help protect our infrastructure software, we modified this algorithm to kill DECREE processes first, before any native processes are considered.

Linux has two `mmap()` implementations, the default and the "legacy" `mmap()`, the latter begins allocating pages at 0x40000000 and proceeds upward. The latter starts below the minimum stack address (0xba2ab000 for DECREE processes) and works it way down as pages are allocated. The choice for the method used is based on the stack limit for the process. Our DECREE loader ensures that the stack limit of 8MB is in effect (this limit would normally be inherited by a process from its parent). We noticed this behavior by unit testing `allocate()` and observing different results when a DECREE binary was run under GNUmake (which sets a very large stack limit and thus uses the "legacy" `mmap()` implementation) and Python which uses the default `mmap()`.

The `wait4()` system call returns information about the resources used by an exiting process. Our process monitoring utilities make use of `wait4()` to gather performance measurements on DECREE processes. We observed that the maximum resident set size (MAXRSS) value returned via `wait4()` was incorrect for DECREE binaries. The MAXRSS value is a measure of the maximum number of pages resident in main memory for a given process. The value was being inherited by DECREE processes from their native parents. Thus trivial DECREE binaries consisting of only a small number of pages reflected a MAXRSS value of a larger parent process. The DECREE loader ensures that the initial MAXRSS is set to zero and does similarly for the count of minor faults (MINFLT).

3.4 FreeBSD Implementation

For the CGC Final Event (CFE) DECREE was ported to FreeBSD 10.2 running with under a custom written "blue-pill" style hypervisor. The CPU was chosen in 2014 (Intel Haswell micro-architecture) and our competitors expected us to keep the same environment for CFE (August 2016) as we provided in the qualifying Event over a year earlier (June 2015).

Implementation of our DECREE loader and system calls is similar to that of Linux. The modified version of the ELF loader forms the base and DECREE-specific hooks are added. The loader for FreeBSD is implemented as a loadable kernel module (LKM). The loader also sets the behavior of signals which allows for mapping FreeBSD signals to the Linux equivalents.

A hypervisor is required to support several of the consistency measures described in Section 3. A custom hypervisor, implemented as a LKM was created. For performance sake, the hypervisor disables all VM Exits not required for instruction emulation. For example, by default nearly all accesses to MSRs cause VM Exits. Haswell micro-architecture processors support a bit mask of MSRs readable and or writeable without hypervisor interaction. The CFE hypervisor explicitly allows read and write access to the performance monitoring subsystem of the processor without hypervisor intervention.

One of the primary difficulties in porting DECREE to FreeBSD was implementation of the virtual address allocation algorithm used by `allocate()` (implemented in terms of Linux's `mmap()` system call). For the port to FreeBSD we had to guarantee the same addresses would be generated for each call to `allocate()` in both DECREE under Linux and DECREE under FreeBSD. This was solved by lifting the virtual address allocation algorithm directly out of Linux. A shadow Linux memory management structure (mm) is generated from the FreeBSD virtual memory map (vmmap) when needed. The Linux algorithm is called and a virtual address is produced. This virtual address is then used to force the FreeBSD vmmap to contain the same address (using `mmap()` with MAP_FIXED).

Several problems were encountered with memory performance measurement consistency between Linux and FreeBSD. By default Linux does not prefault pages; each page is faulted individually and thus the minor fault count follows each initial page access. Prefaulting is default behavior on FreeBSD however. For example, when a program is loaded, several text pages are loaded when possible during each fault. The result is that the minor fault count no longer follows the number of accessed pages. Fortunately, it was possible to disable the prefaulting behavior for DECREE binaries under FreeBSD.

Virtual memory accounting consistency also required

careful examination of the maximum resident set size. FreeBSD counts pages used in page tables, page directories, etc. as part of the resident set size of the process; Linux does not. The virtual memory system of FreeBSD was augmented to track the over-count and subtract it when reporting the maximum resident set size.

Linux and FreeBSD both automatically add pages to the stack segment of a process when a fault occurs near the stack pointer. However, FreeBSD allocates virtual space for the entire stack space when a process is started; any fault within this range causes a zero-filled page to be put in place. Linux adds an additional check on the distance between the stack pointer register and the faulting address. If the address is more than 65664 bytes below the stack pointer, the process is sent a segmentation fault signal [14]. This distance check prevents Linux processes from reading and writing arbitrary addresses within their own stack segment. The Linux behavior was ported FreeBSD and applied only to DECREE processes.

We also encountered system crashes in the performance monitor counter (PMC) subsystem in FreeBSD. It is believed that these crashes may be fixed in FreeBSD 10.3 and greater, but our workaround was to replace the PMC subsystem with a simplified and purpose-built implementation called XPMC. This subsystem assumes it is on an Intel Haswell and that the required performance counters are available.

The last major hurdle to DECREE on FreeBSD was ensuring system call errors were checked in the same order and returned identical values on both operating systems. For instance, Linux system call handlers call `access_ok()` to determine whether addresses can be read or written. Calls to `access_ok()` are computationally inexpensive and done early in the handlers for all pointers (essentially they only check to see if the address is in the kernel virtual address space and returns an error if they are). FreeBSD, on the other hand, checks for valid file descriptors and other errors and delays the address validity check to the point at which data is actually copied from or to the process by the kernel (it depends on the resulting fault to detect the error). We followed the logic through each system call and rearranged checks to ensure error checking produced consistent results.

4 Discussion and Related Work

The seven system calls specified by DECREE could have been implemented using system call policy enforcement such as systrace [17] or SECCOMP-BPF [6]). However, implementation as a loader and alternative system call table, provides a more easily audited and ultimately more portable environment. We demonstrated this portability by implementing DECREE on both Linux and FreeBSD.

In [7; 8; 3; 2], various methods of detecting hypervisors are discussed. Our goal is not to be an undetectable hypervisor; instead, the goal is to be an extremely consistent environment regardless of the native host operating system. However, these works did influence the decision to ensure local sources of timing information are not available to DECREE programs (e.g. RDTSC, RDTSCP, and `fdwait()`). By extension, similar sources of external entropy from the performance monitoring counters (RDPMC) and random number generator (RDRAND and RDSEED) are also disabled.

In [20], the need to intercept the instructions mentioned in Section 3.2 is discussed. At the time of its writing there was no way of guarding against the execution of sensitive unprivileged instructions (SGDT, SIDT, SLDT, and STR) except by system emulation or translation. The performance impact and risk of divergent behavior was deemed too high for emulation or translation. Support for descriptor table exiting was added to Intel Virtualization Technology (VT-x) processors in Nehalem and Westmere in 2009 [9]. We added support to Xen/QEMU and our own hypervisor to handle descriptor table instruction interception.

Both [22] and [10] describe hypervisors as a part of root kits where the hypervisor is loaded by the host operating system and becomes a parasitic part of the operating system. More recently, [4] discusses a symbiotic hypervisor that provides services to the guest operating system. This style of hypervisor is implemented for the FreeBSD/AMD64 platform used to run the CFE. This hypervisor allows us to intercept various instructions and either signal the host operating system to kill the process (e.g. RDRAND) or to return false but constant data to the process (e.g. CPUID requests).

Attacks on hypervisors by guest processes such as those described in [15; 16] were considered as part of the threat model for CGC. Since we control the guest operating system, the attack surface on the hypervisor is limited to arbitrary unprivileged user-space programs. Further, the number of entry points from those programs to the kernel is severely limited (page faults and the 7 system calls).

Competitors in the Cyber Grand Challenge submitted binaries for use in CFE and opposing teams could examine all submissions. One competitor took advantage of the fact that our competition environment always returned the same values for CPUID. When run, the binary checks to see if it is running in the final event DECREE environment. If so, the binary behaves normally answering polls, etc. If not, the code path exhibited by the binary leads to a simple exploitable buffer overflow. Opposition teams not correctly emulating the CPUID instruction analyzed the binary and incorrectly assumed the buffer overflow was reachable in the CFE environ-

```
xor %eax, %eax
inc %eax
push %eax
push %eax
push %eax
fldt (%esp)
fsqrt
```

Figure 2: Adversarial replacement: FSQRT

ment.

This highlights the problem of perfectly modeling a processor: IA-32 consists of hundreds of instructions and modeling all of them perfectly is difficult. Another team submitted binaries that began with the code in Figure 2 which takes advantage of an emulation bug in QEMU for the FSQRT instruction [23; 24; 13]. Executed on physical hardware, FSQRT of a "unnormal" number results in a non-signaling "not a number". Executed under QEMU emulation, the instruction results in an infinite loop. Any rival using unpatched QEMU would have fallen prey to this emulation bug. It is not known whether any rival teams were greatly affected by this bug.

The CGC Final Event (CFE) ran on bare metal computers with our custom hypervisor and general purpose operating system. It is possible to properly emulate our environment at least to guarantee that the methods used by CGC competitors to cause execution divergence are ineffective. The custom, special-purpose forensic system used to monitor behavior of competitor binaries produced identical results to the CFE environment. Further, a test environment was run with Linux as the host operating system and VMWare Workstation as the root hypervisor. Our hypervisor and DECREE under FreeBSD were running with nested virtualization enabled and we observed identical behavior to the CFE environment for all competitor-submitted binaries tested. In truth, this was the authors' development environment since access to the CFE production equipment was time shared for several purposes leading up to the CFE.

5 Conclusion

For the DARPA Cyber Grand Challenge (CGC) Qualifying Event (CQE), DECREE was implemented under a modified Xen Hypervisor with a Linux host and a Linux guest. Using the modified hypervisor and guest kernel, we are able to make the behavior of many instructions reproducible in other environments.

For the CGC Final Event (CFE), performance requirements dictated use of bare-metal servers, and DECREE was implemented using a custom-written hypervisor on a modified 64-bit FreeBSD. Because of the work done for CQE, it was possible to make the view presented to DECREE challenge binaries in CFE consistent with the environment presented during the previous milestones in CGC. This paper describes the methods by which that consistency is achieved, ultimately making one implementation indistinguishable from the other. In fact, some CGC competitors took advantage of the consistency of the environment to differentiate between the CFE environment and environments used by other competitors for automated binary analysis (e.g. virtualized or emulated).

By carefully limiting the instructions and operating system interfaces available to challenge binaries, DECREE limits the amount of external entropy available to a process. We enumerate external sources of entropy and detail how user processes are prevented from making used of each. The determinism created by limited external entropy paired with system call monitoring in the DECREE environment should be sufficient to reproduce the branch decisions of challenge binaries executed in other DECREE implementations.

Ultimately, our quite disparate implementations of DECREE have already been employed in two high-profile, public experiments: CQE and CFE. Moreover, our implementations have thus far proven to satisfy the properties of determinism and indistinguishability.

6 Acknowledgements

This work was sponsored by the Defense Advanced Research Projects Agency under Air Force Contract # FA8750-12-D-0005. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

7 Availability

The source code for the Linux kernel (including the loader and modifications described), patch for the Xen hypervisor, patch for FreeBSD 10.2, FreeBSD loader module, FreeBSD performance measurement module, and FreeBSD hypervisor are available on:

`https://github.com/CyberGrandChallenge`

References

[1] 1997. System V Application Binary Interface. (March 1997). `http://www.sco.com/developers/devspecs/gabi41.pdf`

[2] 2012. *Hypervisor Top-Level Functional Specification: Windows Server 2012* (version 3.0a ed.). Microsoft, Chapter Feature and Interface Discovery, 5–13.

[3] 2015. Mechanisms to determine if software is running in a VMware virtual machine (1009458). (January 2015). `http://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=1009458`

[4] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *IEEE Symposium on Security and Privacy*. IEEE, San Jose, CA, 763–780. DOI:http://dx.doi.org/10.1109/SP.2015.52

[5] DARPA. 2014. Cyber Grand Challenge Rules, Version 3. (18 Nov 2014). http://archive.darpa.mil/CyberGrandChallenge_CompetitorSite/Files/CGC_Rules_18_Nov_14_Version_3.pdf

[6] Will Drewry. 2012. Seccomp filtering. Linux Kernel documentation. (Jan 2012). https://lwn.net/Articles/475049/

[7] Peter Ferrie. 2006. *Attacks on Virtual Machine Emulators*. Technical Report. Symantec. https://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf

[8] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. 2007. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *11th Workshop on Hot Topics in Operating Systems (HotOS-XI)*. USENIX.

[9] Intel 2009. *A Primer On Virtualization*. Intel. https://software.intel.com/sites/default/files/m/1/d/5/7/f/26677-A-Primer-on-Virtualization.pptx

[10] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. 2006. SubVirt: Implementing malware with virtual machines. In *Symposium on Security and Privacy*. IEEE, 314–327.

[11] Tobias Klein. 2003. scoopy doo - VMware Fingerprint Suite. (2003). https://web.archive.org/web/20070102213035/http://www.trapkit.de/research/vmm/scoopydoo/README

[12] Tobias Klein. 2008. ScoopyNG - The VMware detection tool. (2008). http://www.trapkit.de/research/vmm/scoopyng/index.html

[13] Peter Maydell. 2016. Hang in fsqrt. (July 2016). https://bugs.launchpad.net/qemu/+bug/1603734

[14] Ingo Molnar. 2008. x86: last of trivial fault_32—64.c unification. (January 2008). https://github.com/torvalds/linux/commit/6f4d368ef9e9f91aa0019c11e90773ea32d94625

[15] Tavis Ormandy. 2007. An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments. (January 2007). http://taviso.decsystem.org/virtsec.pdf

[16] Gabor Pek, Levente Buttyan, and Boldizsar Bencsath. 2013. A Survey of Security Issues in Hardware Virtualization. *Computing Surverys* 45, 3 (July 2013), 40:1–40:34. DOI:http://dx.doi.org/10.1145/2480741.2480757

[17] Niels Provos. 2003. Improving Host Security with System Call Policies. In *12th USENIX Security Symposium,*. Washington DC.

[18] Danny Quist and Val Smith. *Detecting the Presence of Virtual Machines Using the Local Data Table*. Technical Report. Offensive Computing.

[19] Danny Quist and Val Smith. 2006. Further Down the VM Spiral: Detection of full and partial emulation for IA-32 virtual machines. In *DEFCON 14*. https://dl.packetstormsecurity.net/papers/general/dquist_valsmith_further_down_the_vm_spiral.pdf

[20] John Scott Robin and Cynthia E. Irvine. 2000. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *9th Annual Security Symposium*. USENIX, 129–144.

[21] Joanna Rutkowska. 2004. Red Pill... or how to detect VMM using (almost) one CPU instruction. (November 2004). https://web.archive.org/web/20041130172213/http://invisiblethings.org/papers/redpill.html

[22] Joanna Rutkowska and Alexander Tereshkin. 2007. IsGameOver(), anyone? (August 2007). http://invisiblethingslab.com/resources/bh07/IsGameOver.pdf

[23] Giovanni Vigna and Christopher Kruegel. 2016. Cyber Grand Shellphish. (August 2016). https://media.defcon.org/DEF\%20CON\%2024/DEF\%20CON\%2024\%20presentations/DEFCON-24-Shellphish-Cyber\%20Grand\%20Shellphish-UPDATED.pdf

[24] Zardus, mike_pizza, anton00b, salls, fish, nobhiros, cao, donfos, hacopo, nezorg, rhelmot, paul, and zanardi. 2017. Cyber Grand Shellphish. *Phrack* (2017). http://phrack.org/papers/cyber_grand_shellphish.html