# BP: DECREE: A Platform and Benchmark Corpus for Repeatable and Reproducible Security Experiments

The DECREE Team

***Abstract*—Reproducibility is one of the core tenets of science. Researchers must be able to independently verify scientific theories and research results; practitioners need to able to trust test results and assessments. Reproducibility is hard because computing environments can exhibit non-deterministic and even diverging behaviors due to differing hardware and software configurations. Software running on one platform might behave differently when executed on another. One might think that constraining configurations using virtualization is the answer, but even machine instructions can be sources of non-determinism. The challenges, trade-offs, lessons learned and our solutions (e.g., use a restricted instruction set) in creating DECREE, a repeatable and reproducible computing environment are detailed in this paper.**

## I. Introduction

Security experiments, like scientific ones, are designed to test specific hypotheses and require variables to be carefully partitioned into ones that need to be tested and ones that do not. Then sound arguments on why the variables not being tested can in fact be ignored must also be established. Once an experiment has been designed, performed, data gathered, and analyzed, researchers then publish the results along with details of the actual experimental setup so that independent researchers can verify the veracity of claims by repeating and/or reproducing the experiment and results [28].

Repeating the experiment serves to demonstrate the trustworthiness of results and reproducing the experiment serves to demonstrate the validity of novel ideas and innovative claims. Therefore, repeatability and reproducibility are foundational to science and security.

In a 2013 study [7], Collberg et al. found that only 217 out of 402 then recent papers published in respected conferences and journals were *weakly repeatable*. The authors further pointed out that they were unable to build 23 of the 217 systems and relied on the original paper authors to assure them that the published system indeed works. The main question to be answered is what happened to this 10% of samples? Why did the software successfully build on the authors systems but not on Collberg et al.'s systems? This paper answers this question indirectly by shedding light onto the problem of repeatability and reproducibility in practical software systems as well as explains how we were able to overcome the challenges.

Our overall goal was to create a computing environment where competitors can participate in a Capture The Flag (CTF) competition to find vulnerabilities in raw x86 binaries and crash the software by leveraging them [3]. Because it is a competition, repeatability and reproducibility are paramount, otherwise the final rankings aren't dependable. It is important to note that repeatability and reproducibility are also our security requirements. Our secondary goal was to create a system that the vulnerability, program analysis, and software testing research communities can use to perform their scientific studies.

This paper's main contributions to the community are an experimental platform called DECREE, a corpus of 131 test applications with built-in memory corruption vulnerabilities that are repeatable and reproducible and the documentation of lessons learned. In fact, DECREE is already used for some research efforts [5], [25], [26].

DECREE (DARPA Experimental Cyber Research Evaluation Environment), is a system that supports a subset of the 32-bit x86 instruction set architecture, a single toolchain, a simple application binary interface and a limited interprocess communications model. These design decisions, while restrictive, were necessary to ensure that DECREE and its results are repeatable and reproducible. DECREE is not perfect for everyone, but we hope that this paper can help other researchers design and implement more advanced systems to suit their needs.

We discuss three main lessons learned:

1) Complexity introduces unknowns and uncertainties which reduce trustworthiness. We had to systematically reduced complexity to the bare minimum that is needed for the competition. Complexity is also detrimental to researchers since complex systems are more difficult to analyze and specific variables separated for testing. This in turn, reduces the veracity of security claims.

2) Repeatability necessitates determinism; what is not deterministic cannot be repeatable, but determinism is not sufficient for repeatability. We systematically identified pockets of determinism in modern computing environments that we can rely on or that we can create tools to control. All others are simply unsupported.

3) Repeatability necessitates automation; human errors are inevitable and human knowledge is inevitably lost. Thus, we have also created a toolchain that automates the process of configuring, building and testing the applications for deterministic behavior. A user simply runs `make`; the rest is fully automated. This minimizes human induced non-determinism.

For the research community, these lessons learned can be

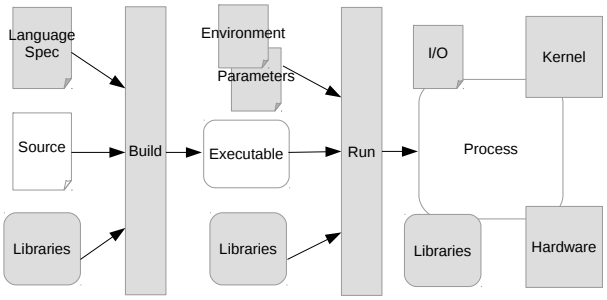


Fig. 1: A basic depiction of all external variables (in gray) that affects the different stages of a program's lifecycle (in white).

summarized as: the cost/benefit tradeoff of using real-world applications for experimentation must be carefully considered and clearly documented. We present DECREE as an experimentation platform for research where the added complexities and non-deterministic behaviors of real-world software is unwarrented and even detrimental. For example DECREE can be used for early on research, as an intial gate that must be passed prior to testing on real-world software, etc.

The rest of the paper is organized as follows. Background information on the problems of abstraction and determinism are presented in Section II. The design of DECREE and the corpus are presented in Section III and their implementation in Section IV. Finally the limitations and lessons learned are presented in Section V and conclusions drawn in Section VI.

Please note that in the interest of brevity we will use repeatability, reproducibility and dependability interchangeably to mean all three concepts unless specified otherwise. Furthermore, this paper is written for a scientific research audience instead of competition, scientific, practitioner, etc.. For example, when we discuss researchers conducting experiments, we also mean competitors solving challenges.

## II. Background and Motivations

### A. Complexity

Modern computing architectures are complex. At minimum there are two abstraction layers, hardware and software, but in practice there are many more. The hardware can be separated into the different components each with its own processor and firmware. Even processors can be separated into different families and models each with their own features, extensions and errata. Then, software can be separated into the privileged operating system which further abstracts away details such as how to access network devices and file systems, and the unprivileged userspace applications which depend on shared libraries either statically or dynamically linked.

The relationship between complexity and repeatability is notionally depicted in Figure 1. A software tool is often the realization of the novel idea presented in systems and security research papers. This software is then shared with the community either as source code on the far left of the figure or as an executable in the middle. Similarly, a competition challenge is shared amongst all of the competitors. In either case experimental or competition results are only generated by executing the executable with the chosen data set as depicted on the right. However, the repeatable execution of the binary now depends on the loader that "runs" the executable as well as the rest of the system such as environment settings, shared libraries and kernel settings.

The external dependencies depicted in gray are numerous but still incomplete. In reality, the system used to perform one experiment is likely different from the system used to perform another. This is evidenced by the difficulties faced by Collberg et al. when trying to simply repeat the building process from source to executable.

Complex systems are not only difficult to replicate, but they can also hinder scientific progress. For example implementing reference monitors at the system call level is known to be difficult due to the complexity of system calls and also abstractions made within the Linux kernel itself [12]. As an additional example, Qu et al. found that the effectiveness of concolic testing tools such as KLEE was greatly limited by floating point numbers, native calls (e.g., library and system calls) and pointers [24]. Finally a more recent example can be found in DroidSafe [13], where the authors expended significant engineering effort to develop an Android Device Implementation that, while incomplete, models the Android platform sufficiently to reveal complex information flows in representative benchmark programs.

As platform designers, our goals are to limit and control variability - removing the gray boxes - not only for us but also for our users who we hope will use our platform and resulting corpus. Furthermore we seek to do this without oversimplifying the system and rendering the results impractical for real-world applications.

### B. Configurations and Compilers

Yet another dimension to complexity is configurability. This includes the ability for systems to be provisioned with different software versions as well as different configuration options.

Having different configurations can result in drastically different behaviors as evidenced in CVE-2002-1337 [1] where the Sendmail buffer overflow vulnerability can result in arbitrary code execution in some systems but not others despite the fact that the same source code was used to generate the binaries [17]. Similarly, changing the linking order (alphabetical vs. default) while building the source can also drastically alter the performance characteristics of the resulting binary [22].

It is important to also recognize implicit options and undefined behavior. In a 2013 study, Wang et al. showed that vulnerabilities were introduced in software binaries because safety and security checks in the source code were optimized out by compilers at certain optimization levels [29]. In short certain behaviors are left "undefined" in the C specification, and the compilers happened to concretize that undefined behavior in a way (e.g., remove the offending statement) that led to vulnerabilities.

For repeatability, another troubling aspect of the study was how the behavior varied across compiler versions. For

example, the check `if(x + 100 < x)`, where `x` is a signed int is removed with *-O1* in gcc versions 2.95.3 and 3.4.6 but with *-O2* in versions 4.2.1 and 4.8.1. The optimization level was also different for different compilers: clang-1.0 did not remove the check at all and clang-3.3 removed the check with *-O1*.

To put it differently, knowing what the configuration options are might be insufficient for repeatability - one must also ensure that all dependencies have the same versions and are equivalently configured. This observation brings into question the standard use of *Makefiles* that separate the compilation options from the compiler being used.

Different compilers can also generate different machine code, once again affecting repeatability. Floating point calculations are known to be difficult to repeat across different hardware due to different concretizations of undefined behavior in the IEEE 754 standard [23]. The same issues can also surface by using different compilers and compiler targets.

System configuration options such as environment variables are extremely important as well. In the same study that showed the correlation between linking order and performance, Mytkowicz et al. also showed that the environment variable size affects performance since it can alter the alignment of data variables on the stack with cache lines [22]. Security researchers are also familiar with environment variables changing the addresses of stack variables for example.

In the end, the hardware and software configurations must be known or fixed in order to achieve reproducibility. One common solution is to build a virtual machine (VM) and deliver that as a research artifact instead of only providing the source code [15]. A virtual machine is a convenient mechanism for encapsulating the experimental software and its dependencies into a single portable image. This luxury of Computer Science research is already used in a number of vulnerability testing projects where repeatability is paramount such as Metasploitable [21] and SEED [11].

Virtualization is insufficient for the DARPA Cyber Grand Challenge though since competitors are expected to patch vulnerable programs at the binary level meaning there could be vioaltions of the intended layers of abstractions. Furthermore, virtualization is itself an abstraction layer which can in turn introduce challenges to repeatability.

### C. Abstractions

Abstraction [8] is a powerful technique for hiding complexities. The operating system, shared libraries and compilers highlight the benefits of abstractions. A user seeking to write data to a file only needs to make a couple of system calls to open and then write to the file. The details of where the file is located on the physical disk, which bus the disk controller is connected to, how to communicate and synchronize with the hardware, etc. are all handled by the OS.

Abstractions offer great conveniences on the one hand, but can also degrade repeatability on the other. This is especially true for vulnerability analysis experiments since the existence of vulnerabilities and the exploitability of them can depend on how the abstractions were implemented.

The *double-free* class of vulnerabilities is a good example where exploits must be tailored to the dynamic memory allocator such as *dlmalloc*. While the caller to *free* might not notice if an alternative malloc library is being used, a vulnerability analyst and researcher must be cognizant of such changes.

Abstractions can also lead to incorrect assumptions and models. A recent example is CVE-2016-0777 where an SSH client's private key information remains in memory even though the client zeroes out all known temporary copies of the key. As it turns out, the private key was read in using *fgets()* which had its own internal buffer [4]. Since the buffer is internal to libc, it is not exposed to the library user and the user can't directly clear the buffer.

It is also interesting to point out that the vulnerability is also highly dependent on the supporting software being used. That is, the private key data is effectively cleared on Linux implementations of glibc since they use *mmap()* and *munmap()* to create and destroy the buffers. BSD-based systems use *malloc()* and *free()* which doesn't actually zero out the allocated space.

The vulnerability can also be affected by the compilers as zeroing the buffer prior to being *free()*'ed can be optimized out since there are no subsequent reads from the buffers before they are destroyed. This optimization is known as *dead store elimination*. Once again, configuration and compilers matter for repeatability and reproducibility.

Virtualization (to include emulation) is another kind of abstraction that suffers from repeatability and reproducibility issues. Some malware actively try to detect emulated, virtualized and native environments and take different execution paths accordingly for example. Even if the software were benign, differences between real and virtualized hardware devices, timing, and instruction execution behavior (e.g., lazy flags calculation and exception signaling are two large sources of differences between QEMU and x86 [20]) can hamper repeatability.

Similar to the conclusion of the previous section, care must be taken to ensure that abstractions remain consistent across different execution platforms in order to achieve repeatability.

### D. Determinism

Determinism is a natural means to achieve repeatability. Determinism guarantees that the same inputs will generate the same outputs and is thus repeatable.

Record and Replay is one research area that has paid particular attention to the determinism of software applications at runtime. A common theme across record and replay tools is the identification and recording of all sources of non-determinism.

Record and replay tools have been developed at various layers of abstraction from individual applications to entire systems. For example VMware introduced whole system record and replay into VMware Workstation 6 in 2007 [14]. As a

testament to repeatability, users can also "go-live" at any point during a replay to gain control of the virtual machine. As a testament to how difficult the problem is in practice, this feature is no longer supported.

PANDA [10] is a QEMU based analysis platform that supports architecture-neutral whole (software) system record and replay. PANDA records all inputs to the system (i.e., Port IO, DMA and Interrupts) and provides repeatability guarantees for replaying entire software stacks. It also does not support going-live since hardware state information is not recorded or replayed.

At the next levels of abstraction are Arnold [9], where all user-level processes of a system are recorded, and Scribe [16], where a single process and its children can be recorded. Both Scribe and Arnold not only record the target application's execution, but also that of any other applications that it might depend on. To put it differently, they record the execution of a working set of processes in order to repeatably replay execution. Similar in vein to the observations above, understanding dependencies is important for repeatability.

Aftersight [6] and Transplay [27] are good examples to demonstrate reproducibility in record and replay systems; both systems support heterogeneous record and replay. Aftersight takes VMware recordings and replays them in QEMU while Transplay takes its own logs of Linux applications and faithfully replays them in Windows with the help of Intel Pin [19] instrumentation.

## III. Design

Our overall goal is to design a system on which repeatable experiments can be conducted. The lessons learned in repeatability and reproducibility can be readily applied to the design of secure systems since secure systems have similar traits. This section describes the techniques used to address the concerns raised in the previous section.

### A. Design Rules

The key design decisions and rules for DECREE are summarized in Table I. Descriptions of important designs are discussed in the rest of this section.

The table shows that many of our design considerations were targeted towards complexity. Reducing complexity not only improves repeatability and reproducibility, but also reduces the initial cost of entry for using and experimenting with the vulnerable software corpus. This is why reducing complexity must be prioritized. Removing and reducing the need for abstractions was our next priority followed by determinism and finally differences due to configurability.

There are two main ways to reduce complexity: we can start with a clean slate or take a current system and remove complexity. The clean slate approach has the advantage of being more precise (building exactly what is needed and nothing more) but has the disadvantage of higher development costs, while the latter approach has the advantage of being more realistic. We applied both approaches in DECREE.

It is important to note that the subtractive approach requires developers to have a firm understanding of current systems including all underlying abstraction layers. A mistake or misunderstanding can lead to incorrect assumptions that will eventually affect repeatability. We will discuss an example of where an oversight led to a race condition that affected repeatability and how we addressed it in Section V.

*1) DECREE Design:* Hardware configurations can vary widely and modern operating systems must be able to communicate and coordinate with a wide range of them, making the OS itself complex. Developing a new OS is extremely challenging and therefore we avoided it; instead, we built DECREE on top of i386 Linux.

OSes abstract away many of the hardware details and present userspace programs with an extremely simplified execution environment called the Application Binary Interface (ABI) consisting of unprivileged instruction sets from the CPU and a system call interface for interacting with the rest of the system. Therefore the first natural design choice is to leave the kernel alone and focus our efforts on userspace programs. We had two focus areas corresponding to those of the ABI.

**Instruction Sets** The x86 architecture is extremely rich and modern processors include a number of unprivileged instruction sets and extensions including both 32 and 64-bit versions of the original x86 instructions. Additionally new instructions and extensions are introduced regularly. Some of these can reduce repeatability like the ever present `cpuid` instruction used for identifying CPU features.

In the end, we decided to support the default 32-bit x86 and floating point instruction sets used by compilers such as GCC and Clang (e.g., i386-linux-gnu). The integer instructions are important because they are the basis of the x86 instruction set and many software applications rely on them. The floating point instructions are also heavily used by scientific applications and have resulted in infamous bugs (e.g. Ariane 5 rocket failure), and thus should be part of DECREE.

This decision was supported by two major observations. First, these instructions are pervasive across all x86 processors and most of the instructions are either found natively in other architectures or can be easily emulated[1]. Repeatability issues across hardware platforms are therefore mitigated. Second, these instructions are well studied and either well supported by analysis tools or will be in the near future (in the case of floating point) meaning researchers can readily utilize DECREE to support their work.

The only remaining complication is which extensions to support. Compilers are designed to balance portability and performance by default. They do this by first detecting the host's configuration - known as the target triple in Clang, which includes the CPU architecture (e.g., i386), the system (e.g., linux) and the ABI (e.g., gnu) - and then configuring the build process to suit the target. Thus, as long as we don't

[1]While some architectures do not have native floating point support, soft-float implementations such as the one found in QEMU and the Linux kernel are readily available.

TABLE I: High Level Design Rules and Ideas - Comp(lexity), Con(figurations), Abs(tractions), and Det(erminism)

| Constraint | Com | Con | Abs | Det | Notes | Design Rule | Implementation |
|---|---|---|---|---|---|---|---|
| ISA - x86 | X | | | X | Non-deterministic instructions such as `rdtsc`, `rdrand`, etc are disallowed | 32-bit ring 3 | Honor System* |
| ISA - Extensions | X | | X | | x87/MMX/SSE/SSE2 | Floating Point Only | Honor System* |
| Dev - Language | X | | X | | - C/C++ were chosen since they are popular<br>- No dynamically generated code<br>- No inline-assembly | Low-level language | C/C++ |
| Dev - Compiler | | | X | | Clang has a number of open source analysis tools | Single | Clang |
| Dev - Build | X | X | X | X | - The default DECREE Makefile not only builds the binary, but also runs repeatability tests | Make and Test | Makefile |
| Shared Libraries | X | X | X | X | Standard libraries have not been ported, must be built from scratch | Static Only | Binary Format |
| ENV / Arguments | X | X | | X | No environment variables | Disallowed | Binary Format |
| Input/Output | X | | | X | No program arguments | Restricted | *receive()* and *transmit()* |
| State | X | | | X | No file system access and no direct network access | Stateless | System Calls |
| IPC | X | | X | | No shared memory | Message Passing | *receive()* and *transmit()* |
| Entropy | | | | X | A source of entropy is required for realism | Pseudo-Random | *random()* system call |

*`rdtsc`, `rdrand` and other instructions can be restricted using hypervisors, but it is easier to ask developers not to use undesirable features

change the underlying architecture, repeatability should not be an issue.

In our testing of Clang over a wide range of hardware platforms, we have found that it uses a default CPU class of Pentium4 (introduced in 2000). According to the Intel Software Developer's Manual [2], this means that instructions up to SSE2 are supported by default.

As discussed previously, relying on default configurations can be dangerous for repeatability and reproducibility. For that reason, while we allow the compiler to generate the binary in a repeatable manner using any instruction sets it might deem reasonable for an architecture, we also use a tool to verify that only certain instructions were utilized. As an additional check, we also ensure that the same binary is generated by each build run by comparing their MD5 sums.

**Special Instructions** We also exclude certain instructions (opcodes) that are not readily repeatable. One of the most pertinent is `cpuid` since it was introduced with the 486 and is considered part of the i386 architecture. `cpuid` is used to identify the available processor features and therefore returns different data for different processor architectures. Furthermore, emulation and virtualization environments such as QEMU often use the values returned by the host's hardware, meaning heterogeneity is seen through the emulation layer. Virtualization and emulation does not address this problem by default and special care must be taken to ensure consistency across virtualization and emulation layers [30].

We have also identified `rdtsc` (returns the value of the hardware timestamp counter) `rdtsp`, `rdpmc` (returns the value of a performance counter), `rdrand` (a recent instruction that returns a random number) and `rdseed` (an extension to `rdrand` that can be used to provide entropy for seeding pseudo random number generators) in addition to `cpuid`.

**System Calls** There are well over 300 system calls in a modern Linux ($>$ 3.0) kernel; three hundred system calls that a researcher or tool developer must first model prior to performing analysis or accept the additional complexities of the kernel.

Modeling the system calls is not an easy task either. For example, modeling the *write()* call requires knowledge of what the file descriptor (*fd*) parameter actually refers to. This, in turn, requires the model to track calls to other system calls such as *open()* and *dup()* along with all of the different return codes, thus effectively duplicating the kernel's implementation. Like system call interposition [12], system call modeling is extremely complex and error prone. This does not mean that it's impossible, of course. DroidSafe was able to model the Android framework after all. It is simply resource intensive and must be updated for each and every API version.

Thus, we sought to limit complexity by reducing the system call interface to seven essential system calls. The calls are outlined in Table II. We arrived at these 7 system calls using a clean slate based approach. Input and output are the basis of computing and thus we needed two system calls, one for the application to obtain inputs and one for sending outputs to the rest of the system. We also realized that virtual memory is one of the cornerstones of modern computer architectures and thus, we also needed to support the dynamic allocation and deallocation of memory.

Entropy is also important in vulnerability and program analysis and testing in general. Many programs require a source of entropy in order to be realistic and interesting. Entropy is also an important differentiator for static and dynamic analysis techniques since control logic that depends on random input will not be known until runtime making reachability analysis more difficult. In order to facilitate this, we also added a new system call for obtaining random numbers.

Entropy, while enhancing realism, is detrimental to repeatability unless we can repeat the random numbers provided. In other words, we require the use of a pseudo-random number generator. Full control over the random numbers generated

TABLE II: DECREE System Calls

| Syscall | Linux | Notes |
|---|---|---|
| *_terminate()* | *exit()* | Same |
| *transmit()* | *write()* | *fds* are pre-determined |
| *receive()* | *read()* | *fds* are pre-determined |
| *fdwait()* | *select()* | Same |
| *allocate()* | *mmap()* | Can only set RWX |
| *deallocate()* | *munmap()* | Same |
| *random()* | - | Pseudorandom generator |

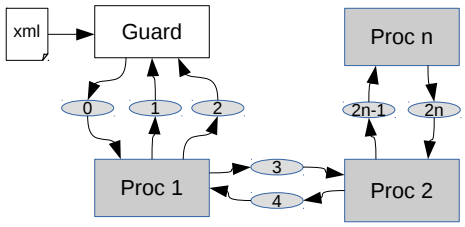


Fig. 2: Communications between benchmark processes (in gray rectangles) take place over predefined fds (in gray ovals). Communications with the output system is mediated by a guard (in white rectangle).

is another reason why we chose to use a system call to provide entropy to a program - we have full control over how the system call will be implemented - instead of a native instruction like `rdrand`.

We added two more system calls, *_terminate()* and *fdwait()* for applications to exit cleanly and also to receive signals of when input is ready for consumption - so the program doesn't have to constantly poll - respectively.

The semantics and function prototypes of DECREE system calls (as shown in Table II) are very similar if not exactly the same as those in Linux. This was done to help reduce the cost of transitioning current research tools and techniques that work for Linux to also work with DECREE and vice versa.

**Inter-Process Communication** The *receive()* and *transmit()* calls are used for input/output. While the function prototypes are similar to those of Linux *read()* and *write()*, the file descriptor parameters *fd* have predefined associations. More precisely, 0 is used for `stdin`, 1 is for `stdout`, 2 is for `stderr` and any higher numbered descriptors are used for inter process communication.

There are two main Inter-Process Communication (IPC) techniques: message passing and shared memory. We decided to support only message passing, because shared memory necessitates mutex support in the kernel (more system calls) or hardware, both of which increase complexity. Furthermore, we decided not to allow the dynamic creation and destruction of IPC channels. They must be pre-allocated.

Figure 2 provides a quick overview of what a multi-process DECREE application looks like. All IPC channels are pre-allocated with reserved file descriptor numbers represented in ovals. In particular, we have reserved 0-2 for the standard input/output that *nix users would expect and a pair of higher numbered *fds* for each additional process beyond the first. That is, if we have n processes then there will be $2n + 1$ file descriptors of which $(2n + 1) - 3$ are used for inter process communication. Preallocating file descriptors is straightforward, as long as we can determine - at load time - how many processes are required.

**Process Management** Creating and destroying processes is a real-world problem with interesting research implications. However, supporting process management requires additional system calls, once again increasing complexity, and also reduces repeatability and reproducibility since it is highly system dependent. For example, the maximum number of processes that can be created at a time can be set in Linux kernels using *ulimit*. Thus a benchmark application that uses many processes might not fail until the ulimit is reached.

In our pre-allocated design, failures due to the lack of resources occur at load time instead of at runtime. We believe that failing early will help increase repeatability of experiments. Creating all processes up front also helps support our simplified IPC design.

**State** Record and replay researchers have found external state to be one of the biggest sources of non-determinism and thus we do not support external state. To put it differently, we do not want repeatability to hinge on needing a complete snapshot of the file system and/or a record of all network traffic. Environment variables are yet another example of external state and therefore are not allowed by design either.

We do not have an *open()* system call, and thus any inputs to benchmark applications must come through `stdin` or from other processes. This does not preclude researchers from implementing a new filesystem interface strictly through `stdin`, of course. Our design discourages this by making it difficult to accomplish.

**Marshalling** We have also introduced a *guard* (see Figure 2) to provide better repeatability guarantees if users choose to utilize it. This guard program marshals and controls all inputs to a DECREE application just as an input replay service would in record and replay.

The guard is designed to read inputs from a structured file (XML) and replay the inputs in order. This design effectively ensures that all inputs to a program must be pre-determined. This, of course, can be overly restrictive since it precludes the authoring of more sophisticated applications that, for example, use challenge and response protocols.

To facilitate creating applications with advanced protocol designs, the guard is also tasked with reading and interpreting output from the application. In this way, the guard can process some output from an application and then re-inject the results as input. However, our guard design restricts output processing to simple operations to minimize the opportunities to introduce non-deterministic inputs to an application.

**Determinism** It is worth mentioning that our design results in a slightly different form of input/output determinism. In particular, since all interactions are mediated by the guard, our only guarantee is that all benchmark programs will produce the same outputs given the same inputs in the same order as described in the structured file to the guard.

Our design does not guarantee that the application itself is fully deterministic, because exhaustive testing is infeasible. Additionally, our design only guarantees that the process (or processes) that reads and writes to `stdin` and `stdout` are input/output deterministic. The other processes in an IPC design are not measured by the guard.

A good example of why only this guarantee can be supported can be found in a simple IPC program with two processes. Suppose process P1 reads an integer $i$ from `stdin` and passes it to process P2. P2 reads the integer $i$ from P1 and calculates $s = r + i$ where $r$ is obtained through the *random()* system call. $s$ is written back to P1. P1 bitwise ANDs $s$ from P2 with $i$ and if the result is odd it will write `ODD` to stdout, otherwise it will write `EVEN` to stdout.

This simple application is deemed deterministic by our definition if all of our input test cases happen to be even, because the output from the application is guaranteed to be `EVEN` if the least significant bit of $i$ is 0 due to the bitwise AND. It is obvious that the entire application is non-deterministic since the output depends on the random integer retrieved from P2.

This example illustrates two important points. First, our design anticipates this type of problem by ensuring that the *random()* system call return pseudo-random numbers. Thus the program is actually deterministic for a particular pseudo-random seed. Second, this highlights the problem that the results are only as good as the tests used to generate them.

*2) Removing Abstractions:* Abstractions hide important details and introduce assumptions, sometimes without the user acknowledging them. Thus if one of these assumptions happen to introduce non-determinism, the overall design can become non-repeatable unless the problem is addressed.

Dennis [8] described three different abstraction techniques: extension, translation and interpretation. The best way to think of an abstraction through extension is a shared library such as *libc*. Shared libraries are usually written in the same programming language or specification as the library's user but it abstracts away many low level implementation details by introducing a simple Application Programmer's Interface. Compilers are good examples of abstraction through translation since the compiler translates a behavior specified in one language, e.g. C, into another, e.g. x86. Finally emulation and interpreted languages such as Python are great examples of abstraction through interpretation.

Of the three types of abstractions, we only need to worry about extension (shared libraries) and translation (compilers), because we expect our benchmark programs to be native x86 binaries.

**Development** As outlined in Table I, we solved the abstractions by translation problem by making the design choice to support only C and C++ programs. We chose C/C++ because they are extremely close in terms of their specifications, they are not type-safe and thus can result in many interesting vulnerabilities, and finally they are not interpreted such as Python and Java thus avoiding having to deal with abstractions through interpretation.

In addition to restricting all benchmark programs be written in C/C++, we further required that they be compiled using a single compiler with a repeatable set of configurations. That is, the configuration for the compiler must be such that the compiler will repeatably generate the same binary given the same source file.

Furthermore, we had to restrict DECREE program authors from using dynamically generated code as well as inline assembly since these are potential sources of non-repeatability. Dynamically generated code represents a different code generation path that we do not control. Not only is this bad for repeatability, but it also creates unnecessary complexity for the test programs.

Inline assembly was also disallowed because we wanted to limit instructions used to generic x86 and floating point only. Allowing instruction inlining can violate this design decision. Furthermore, we believed that allowing inlined assembly can introduce unnecessary complexity, potentially forcing analysis tools to define heuristics to deal with human generated and optimized assembly code. This is undesirable to us, but as we will discuss in the implementation section, this design choice is not strictly enforced thus researchers can always change this if their work deems it necessary.

**Libraries** As depicted in Figure 1, shared libraries can be introduced at all stages of a program's life cycle. Shared libraries are not only complex, but can and are readily replaced with newer or compatible versions in real world systems. This poses a problem to repeatability since, as discussed previously, repeatability can hinge on having the right shared library. This is precisely the case for the recent SSH vulnerability CVE-2016-0777.

To improve repeatability, our design requires all benchmark authors create a new *libc* with a new API. There are two motivations, first this encourages authors to only implement the libc functions they need, thus reducing complexity. Second, because the APIs are different from traditional libraries, one cannot simply replace one shared library with another - particularly the complex libraries found for modern systems.

DECREE also disallows dynamically loaded libraries. In fact, our design requires all benchmark binaries to be statically compiled and linked, making them self-sufficient.

*3) Test and Verify:* Repeatable and reproducible designs are incomplete without the ability to test and verify the claims. In addition to standard unit tests and integration tests that are part of normal development, we have also designed some test and verification functionality into DECREE.

The first major verification-related design choice is to require benchmark authors to effectively implement the benchmark's input/output logic twice. This helps with both repeatability and reproducibility.

Input/output determinism of the application $P$ is cross checked with that of an input/output generator $G$. We are able to do this, because all inputs must pass through the marshalling guard thus requiring all inputs and outputs to be pre-specified in a file. Thus, the second implementation produces the XML file used by the guard to interact with DECREE application.

To mitigate the risk of the input/output generator creating corner cases as described in the trivial IPC program above, we also require the input/output generator to take in, as a parameter, a seed to a random number generator which is then used to generate the XML file. In other words, $G$ will pseudo-randomly generate inputs, calculate the expected outputs and then write them to an XML file. The file is then processed by the guard which verifies that $P$ has the expected input/output behavior.

The second verification-related design choice is to integrate testing into the build process. That is, we define a build as "successful" only if the application binary(ies) is created, the input/output generator has generated enough XML files (e.g. 1,000 interactions), and the binary exhibits the deterministic input/output behavior as defined by the XML files.

The third verification safeguard we have designed into DECREE is to execute the build process multiple times ensuring that the same binary is generated each time (as determined by MD5 sums).

### B. Design Summary

As a quick summary, DECREE's design can be separated into the following philosophies:

1) Reduce complexity: Only support userspace programs. Only support IA-32 and floating point instructions. Do not support for special instructions such as `cpuid`. Simplify the system call interface.
2) Fail early: Pre-load processes. Specify inputs and expected outputs in an XML file. Disallow dynamic libraries.
3) Control external states: Do not allow environment variables. Do not allow arguments. Mediate input and outputs using a guard.
4) Remove abstractions: Only support C/C++. Control compiler configurations. Restrict shared libraries.
5) Trust but verify: Alternate implementation. Make automatically tests and verifies the application. Instruction set verification.

## IV. Implementation

We begin our discussion with the implementation of DECREE followed by a short general description of how the benchmark programs were implemented. Actual statistics and information about the corpus can be found in other documentation such as the open source repository.

### A. DECREE

According to our design principles, we wanted to build a new computing environment that is both repeatable and useful (representative of current systems and compatible with current tools). This was done by creating a new *ELF* compatible binary format on top of a standard i386 Linux distribution - Debian Wheezy with Linux Kernel 3.13.2. This new distribution, which we call DECREE, is therefore compatible with standard Linux i386 binaries (including all of Debian's packages) as well as DECREE binaries.

**DECREE Binary Format** Effectively, DECREE binaries share the same exact file format as a standard *ELF* executable except the file magic `ELF` has been changed to `CGC`. Reusing an existing binary format reduced implementation costs and also makes porting existing tools that can handle ELF binaries to support DECREE binaries simple.

With the new file format, we were also able to reuse code from the ELF loader that is already built into modern Linux kernels. The kernel loader can be quite complicated since it is tasked with parsing the executable file and allocating and loading resources (such as files) as described. Being able to reuse the existing loader reduces development costs, but it can also reduce repeatability since the ELF format is quite rich.

Thus, in lieu of implementing a brand new loader, we re-purposed the logic of the standard ELF loader, but excluded handling of certain sections such as the dynamic loading and symbol sections. In this way, we can readily reuse the existing ELF loader source, while also ensuring repeatability. The `CGC` loader is therefore a simplified version of the ELF loader.

Additionally, our loader ensures that all pages are zeroed by default for repeatability. This means that all slack space as defined by the headers will be 0. The same applies to all dynamically allocated pages using the *allocate()* system call.

**System Calls** Defining a new file format helps differentiate between DECREE binaries and native binaries on the file system, but we also require a technique to differentiate them at runtime because the system call interfaces are completely different.

We overloaded the *personality* field in the *task_struct* structure to help differentiate between normal Linux binaries and DECREE binaries. The basic logic is as follows:

- At load time, we set the process's personality to `CGC`
- When a system call is made, the system call handler checks the personality. If personality is `CGC` then pass control to the DECREE system call handler, else continue processing as normal

To implement DECREE system calls we wrote appropriate wrappers of native Linux system calls (see Table II); one exception was *random()* syscall since we needed to ensure a repeatable source of randomness.

The Linux kernel already has a crypto library, and therefore we associated a new pseudo random number generator with each DECREE process. In particular, we used the "ansi_cprng" that, according to its source file, is based on AES-128.

To seed the PRNG, we re-purposed the program arguments. Since our design does not require program arguments and program arguments are passed to the kernel as part of *execve()*, we simply allowed a seed argument that is processed by the kernel, but never passed to the process.

It is worthwhile to point out that, while we implemented the *allocate()* system call by wrapping Linux's *mmap()*, we also ensured that *mmap()* is always called with `MAP_ANON` and `MAP_PRIVATE` only and mapping addresses cannot be provided. DECREE binaries cannot change these options.

### B. Toolchain

In order to build and debug DECREE binaries, we must have the associated toolchain. Thus far, we have ported the *clang* compiler along with the *llvm* tools such as *opt*, the *ld* linker, *objdump* for disassembly, *readelf* for parsing files, *libopcodes* for assisting with disassembly, etc. We refer interested readers to the source code repository for more information, and will provide a short discussion on *ptrace* support here instead.

Usability is an important design consideration for DECREE and building DECREE on top of Linux helps. This is particularly true because DECREE processes can be *ptraced* just like normal processes. *ptrace* is a *nix system call that can be used by one process to 1) attach to another process, 2) control the execution of the other process and 3) examine and change the state of the tracee process.

*ptrace* is used by popular debugging tools such as *gdb* for online debugging, and *strace* for logging system calls, their parameters and return values. To better support the user, we have also ported these tools to work with DECREE binaries. As a further test of *ptrace* compatibility, we also implemented an instruction tracer that single steps DECREE binaries and logs the instructions executed.

### C. Process Creation

One of the major design considerations was to fail early by pre-loading processes and pre-defining file descriptors. We do this through a userspace program, which we call *cb-server*, that sets up all of the processes in a multi-process binary as well as the IPC channels. Process creation is straight forward since we use the standard *fork()/execve()* method as one would expect from a Linux-based platform.

Communication to and from a benchmark application is not as straightforward since we had to decide on a particular communication abstraction. We decided to use a TCP-based socket for communication instead of the more traditional character devices. Briefly, *cb-server* first creates a TCP socket at a configured port and waits for a connection. Once a connection has been accepted, it will then launch the desired program and redirect the `stdin`, `stdout` file descriptors to the TCP socket. `stderr` is left alone and used for debugging.

We used TCP sockets because we also wanted the ability to mimic real-world network based protocols despite the fact that DECREE benchmark applications cannot access the network. With this implementation choice, users can attach tools such as *tcpdump* to capture the program's communications as network traffic for further analysis. The marshalling guard can behave like a client while the application can act as a server or vice versa.

For Inter-Process Communication, we decided to use a more traditional abstraction called *socketpairs*. *socketpairs* are essentially bi-directional pipes that are created in pairs. This is a good match for our IPC design since we sought to create $2(n-1)$ file descriptors for IPC which equates to $n-1$ socket pairs.

To further simplify implementation, and increase reproducibility, all *fds* are shared by all processes in a multiprocess application. That is, *cb-server* first creates the TCP socket for `stdin` and `stdout` followed by the $n-1$ socketpairs, and then forks and execs all processes that belong to the application.

This implementation choice has the drawback of allowing any one of the processes to read and write the collection of *fds*; however, it was deemed easier for benchmark authors to use the *fds* wisely versus complicating the implementation. For example, to achieve the setup depicted in Figure 2, the program author will have to ensure that Proc1 writes to 3, while Proc2 reads from it, and Proc2 writes to 4 while Proc1 reads from it, etc.

### D. Guard

*cb-test* is our implementation of the guard. In brief, *cb-test* first executes a new instance of *cb-server* to listen to a TCP connection at a specified port. *cb-test* then connects to the server, which triggers the server to launch the benchmark program of choice. Once the connections are made, *cb-test* then reads the inputs and expected outputs from an XML file. Inputs are sent through the TCP connection to the DECREE process until all inputs are exhausted and outputs from the benchmark program are compared with the expected outputs until they are also exhausted. Errors or deviations from the expected output will cause *cb-test* to exit early.

The XML file uses a simple specification that allows users to specify binary and ASCII data to be written to the program and read from the program. Data read from a program can then be either ignored, compared to expected output exactly, partially, or through regular expressions. Portions of the data read from a program can also be stored into temporary variables that can be written back to the program later. We do not currently support processing of variables as discussed previously.

### E. Corpus

We tried to limit the benchmark authors as little as possible. The authors were encouraged to design and implement programs as long as they abided by the design principles described above (e.g., no library reuse, no inlined assembly, etc.) and included a memory corruption vulnerability.

We chose to build memory corruption vulnerabilities directly into the source code as a way to provide benchmarks that are useful for the security community. Since we required vulnerabilities, we also required the authors to provide a patch for the vulnerability and a proof of vulnerability XML file that describes interactions that would crash the vulnerable version of the program but not the patched version.

To automate the testing and verification process, we required the authors to wrap their patches within a preprocessor macro definition such as `PATCHED`. We then provided them with a Makefile that makes two versions of the binary, one with `PATCHED` defined and one without. The Makefile also generates a large number of test files using the alternative implementation as described in the Design section. It also tests the provided proof of vulnerability(ies) on the different versions of the application to ensure that one crashes while

the other doesn't as well as the generated test files pass both the vulnerable and patched binaries.

Finally, we also required authors to provide a README file that describes the program as well as any CWEs that describe the nature of the vulnerability.

### F. Verification

As described in Table I, we rely on the honor system to ensure that certain instructions are not used by the benchmark programs. We made this decision since restricting the instructions will require making changes to the compiler, which will reduce reproducibility and portability. Instead, we implemented a verifier which compares the instructions disassembled using *objdump* with the x86asm.net XML database [18].

Essentially, we identified the instruction group and extension by comparing the raw bytes of each instruction with the x86asm.net database. According to our design rules, we only allowed instructions that belonged to the "generic" and "x87fpu" groups and the "mmx", "sse1" and "sse2" extension groups. We also had to further restrict the instructions as described in the previous section. For example, the verifier will not allow the `cpuid` instruction even though it belongs in the "generic" group.

### G. Implementation Summary

In summary, we implemented DECREE by introducing a new ELF-like binary format into a modern Linux distribution (Debian Wheezy). This reduced development costs while also ensuring compatibility with analysis tools and techniques. We also ported a number of build tools including the *clang* compiler and *llvm* infrastructure as well as *binutils*, *gdb* and *strace*. We also implemented `stdin` and `stdout` of benchmark programs as TCP connections to support users who seek network traffic. *cb-server* and *cb-test* are then used to manage the creation and testing of the benchmark programs. All of this is enclosed within a Virtual Machine image that helps ensure repeatability. We encourage users to download and try it out.

## V. Limitations and Discussion

Our design had to be restrictive in order to ensure repeatability and reproducibility. Thus there are many real world programs, such as parent-child programs, that cannot be implemented in DECREE. This is a major limitation that should be addressed in the future. Despite this limitation, we believe that DECREE provides a solid foundation, and we encourage other researchers to extend it as needed.

We also cannot guarantee that our design is perfectly repeatable, because our testing cannot be exhaustive. As an example, we accidentally found a race condition that made binaries non-repeatable. The root cause of which is our choice of using TCP as the communications protocol between the *cb-test* and the program. In other words, we failed to identify all of the implied assumptions associated with the TCP abstraction during both design and implementation.

The TCP protocol enforces a particular maximum size and kernels normally buffer TCP traffic to improve performance. This internal buffering means that the boundaries between writes from *cb-test* to the program are blurred. As a contrived example, if the XML file specifies that *cb-test* should conduct two writes of 64 bytes each, the receiving kernel might coalesce the two 64 byte writes together due to buffering. Therefore, a benchmark program that reads 128 bytes into an internal buffer of 64 bytes, will result in a buffer overflow if the writes have been coalesced (128 bytes will be read into the buffer) and will not result in a buffer overflow if the read occurred before the two writes were combined (only 64 bytes are read in). The same applies to large writes that might be split into multiple chunks so a read for 128 bytes may or may not result in 128 bytes depending on the state of the buffer.

In the end, we corrected this issue by further requiring the benchmark program authors to use a *read_until()* design versus the less deterministic *receive()* call. We do not preclude other surprises from surfacing.

## VI. Conclusions

In conclusion, we have carefully designed and implemented DECREE, a new operating system for repeatably building and testing programs. We have also created a collection of 131 network services with embedded memory corruption vulnerabilities that can serve as benchmarks for researchers to compare their tools and also to facilitate future research. Both DECREE and its corpus has been demonstrated to be repeatable in our testing. However, our testing was not and could not have been exhaustive. Despite our efforts and design trade-offs, surprises still await. It is out hope that, researchers, competitors, and educators can benefit from DECREE and its test corpus, both have been open sourced, as well as the lessons learned in creating DECREE as documented in this paper. The main takeaway should be the question of whether the functionality desired warrants the additional complexity. In other words, DECREE is limited, yes, but the purposeful limitations removed many of the barriers against reproducibility, repeatability and therefore security.

## Availability

DECREE is open source and is available on github at `https://github.com/CyberGrandChallenge/`. We welcome and encourage researchers to use and contribute to DECREE.

## References


[1] "CVE-2002-1337," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-1337.

[2] "Intel 64 and IA-32 Architectures Software Developer's Manual: Combined Volues: 1, 2A, 2B, 2C, 3A, 3B, 3C and 3D," December 2015, http://www.intel.com/content/dam/www/public/us/ en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[3] "Darpa — cyber grand challenge," November 2016, https://www.cybergrandchallenge.com.

[4] "Roaming through the openssh client: Cve-2016-0777 and cve-2016-0778," January 2016, https://www.qualys.com/2016/01/14/cve-2016-0777-cve-2016-0778/openssh-cve-2016-0777-cve-2016-0778.txt.

[5] "Your tool works better than min? prove it." August 2016, https://blog.trailofbits.com/2016/08/01/your-tool-works-better-than-mine-prove-it/.

[6] J. Chow, T. Garfinkel, and P. M. Chen, "Decoupling dynamic program analysis from execution in virtual environments," in *USENIX 2008 Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2008, pp. 1–14.

[7] C. Collberg, T. Proebsting, and A. M. Warren, "Repeatability and benefaction in computer systems research - a study and a modest proposal," University of Arizona, Tech. Rep. TR 14-04, February 2015.

[8] J. B. Dennis, "The design and construction of software systems," in *Software Engineering, An Advanced Course, Reprint of the First Edition [February 21 - March 3, 1972]*. London, UK, UK: Springer-Verlag, 1975.

[9] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen, "Eidetic systems," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2014, pp. 525–540.

[10] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable reverse engineering for greater good with panda," Columbia University, Tech. Rep. TR CUCS-023-14, February 2014.

[11] W. Du and R. Wang, "Seed: A suite of instructional laboratories for computer security education," *J. Educ. Resour. Comput.*, vol. 8, no. 1, pp. 3:1–3:24, Mar. 2008.

[12] T. Garfinkel, "Traps and pitfalls: Practical problems in system call interposition based security tools," in *In Proc. Network and Distributed Systems Security Symposium*, 2003, pp. 163–176.

[13] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*, 2015.

[14] S. Herrod, "The amazing VM record/replay feature in VMware workstation 6," April 2007, https://blogs.vmware.com/cto/the-amazing-vm-recordreplay-feature-in-vmware-workstation-6/.

[15] S. Krishnamurthi, "Artifact evaluation for software conferences," 2015, http://www.artifact-eval.org/.

[16] O. Laadan, N. Viennot, and J. Nieh, "Transparent, lightweight application execution replay on commodity multiprocessor operating systems," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. New York, NY, USA: ACM, 2010, pp. 155–166.

[17] Last Stage of Delirium, "Technical analysis of the remote sendmail vulnerability," March 2003.

[18] K. Lejska, "X86 opcode and instruction reference," January 2016, http://ref.x86asm.net.

[19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2005, pp. 190–200.

[20] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: Hi-fi tests for lo-fi emulators," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2012, pp. 337–348.

[21] H. D. Moore, "Metasploitable 2 exploitability guide," 2013, https://community.rapid7.com/docs/DOC-1875.

[22] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2009, pp. 265–276.

[23] Oracle, *Oracle Solaris Studio 12.4: Numerical Computation Guide*, January 2015, ch. D, pp. 155–168.

[24] X. Qu and B. Robinson, "A case study of concolic testing tools and their limitations," in *Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 117–126.

[25] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[26] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[27] D. Subhraveti and J. Nieh, "Record and transplay: Partial checkpointing for replay debugging across heterogeneous systems," in *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*. New York, NY, USA: ACM, 2011, pp. 109–120.

[28] J. Vitek, "The case for the three r's of systems research: Repeatability, reproducibility and rigor," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. New York, NY, USA: ACM, 2014, pp. 115–116.

[29] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama, "Towards optimization-safe systems: Analyzing the impact of undefined behavior," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 2013.

[30] J. Wright, "Where am i? operating system and virtualization identification without system calls," in *Proceedings of the Cyber Security Symposium*, 2017.