Proceedings of the 2018 Winter Simulation Conference
M. Rabe, A. A. Juan, N. Mustafee, A. Skoogh, S. Jain, and B. Johansson, eds.

ON REPEATABLE EMULATION IN VIRTUAL TESTBEDS

Vignesh Babu
David M. Nicol

Department of Electrical and Computer Engineering
University of Illinois, Urbana Champaign
1308 W Main St.
Urbana IL-61820, USA

ABSTRACT

Virtual testbeds are essential tools for evaluation of cyber-security problems and cyber-security solutions in embedded control systems such as those that control the power grid. However, without explicit control, virtual testbed behavior is not repeatable, which limits our ability to replay a testbed experiment to re-create a surprising event, or even isolate and fix a bug. In this paper we consider extensions to TimeKeeper emulation framework aimed at improving its repeatability. One approach steps through assembly language instructions, the other asks emulated applications to report their own progress through virtual time. Empirical data demonstrates the potential of our proposals.

1 INTRODUCTION

Evaluation of cyber-security in an embedded system is challenging. Most often the system of interest is on-line and its operators are hesitant to allow anything on the system that might perturb it. This is particularly true when the system controls a critical infrastructure, such as a component of the electric power grid. Still, there are very real needs to evaluate the vulnerability of the system and the need to evaluate cyber-security solutions (both for effectiveness and impact they have on the system's ability to meet real-time response objectives.) A practical approach to meeting these needs is to do as much evaluation as is possible in a testbed. This has its own challenges though, because the number of devices in a deployed system is considerably larger than that in a testbed, and the testbed will not have the same communication infrastructure. This challenge is met though by developing and using *virtual testbeds* which combine real devices, emulated devices, emulated networks, and simulated devices and networks. The work reported in this paper is motivated by our efforts to improve the capability of virtual testbeds to assess cyber-security in embedded systems used to control the power grid.

This paper focuses on an important attribute of a virtual testbed, *repeatability*. Consider—an experiment may reveal that a device's software is vulnerable to a particular combination of events and network conditions, or may exhibit some surprising behavior that is not yet understood, or may simply crash. An ability to repeat or replay the sequence of events and conditions that lead to a state of interest would be extremely valuable, but is not typically available. Lack of repeatable behavior also increases the work needed to compute statistically significant estimates of average behavior. If the testbed does not behave exactly the same way every time given the same initial state and precisely the same events it means that there is some internal variance introduced by the testbed itself. We can think of this variance as "noise"—the larger the noise relative to the signal, the more repetitions are needed to filter through the noise to get to the behaviours of interest.

Network emulation is a powerful tool in the toolbox available for constructing a virtual testbed. Popular network emulators like CORE (Ahrenholz et al. 2008), EMANE (NRL 2010), Mininet (ONF 2016) can execute entire software stacks and include ways to impose constraints on emulated networks through link

delays, bandwidths and flow control mechanisms. However, when enforcing these constraints it is difficult to cause each exchanged message in the emulated system to behave or have the same timing as it would in a real system. For instance, to enforce link delays on individual packets, a network emulator would use timers to wait until the specified duration elapses before forwarding the packet. In such "best-effort" emulations there will be variance in precisely when a timer firing is recognized, depending on what is happening in the operating system or other applications. Furthermore, any computation that references time (for instance, to sleep for a specified interval) will reference the hardware clock of the virtual machine manager; unlike the real system, the virtual machine will sometimes be executing and sometimes not, which means that measured elapsed time in a virtual machine may include periods when it was not running at all. These limitations are exposed when the emulated network models are larger and can yield experimental results which deviate significantly from what would be observed in the real deployment.

These issues can be addressed by giving every emulated node its own *virtual* clock. Any explicit or implicit reference its code makes to time is intercepted and interpreted using the virtual clock as a basis. Periods of real time when a node's emulation is not executing are periods when its virtual clock is not advancing, so measured epochs of virtual time are not skewed by inactivity due to scheduling. Furthermore, when every node has its own notion of virtual time, their executions can be explicitly scheduled to advance them all together in virtual time.

We can create the illusion of concurrency such as seen in the embedded system by maintaining and advancing a separate virtual clock for each node, and causing all timing measurements and timing-influenced aspects of applications running on the node to be tied to it. Incorporation of virtual time (VT) in emulations has a rich history. Gupta et. al (Gupta et al. 2005) introduced the concept of *Time Dilation* which simply re-scaled time returned by the system clock with a multiplicative factor (α) called Time Dilation Factor (TDF). Inside a VM with TDF α, the perception of time for all processes progressed α times slower than real time.

Later, virtual time was extended to OpenVZ container based emulation (Zheng et al. 2012) and to individual processes in the Linux Kernel - TimeKeeper (Lamps et al. 2014). In TimeKeeper, the processes are advanced together through a window of virtual time, like time-stepping through virtual time with the window length defining the step size t. No process is run in the next window before every process has run in the current window. The idea is to keep the virtual clocks of all processes with t units of each other. The smaller is t, the tighter the synchrony. The TimeKeeper scheduler allows a process to advance t units in virtual time, after which control is relinquished back to the scheduler. TimeKeeper allows each process to have its own TDF. Versions of TimeKeeper prior to that reported in this paper used Unix signals and timers to govern scheduling. For a window size t, to run a process with TDF α the TimeKeeper scheduler sends it a Unix signal, and schedules a timer to fire in $\alpha * t$ time units of wallclock time, upon whose firing it sends a Unix signal to stop. Repeatability is an issue though with the non-determinism of these timer firings, leading to the work we report here.

For repeatability we need a new control mechanism which, given a particular starting state of an application and window size t, will for every repetition stop the application in the same state upon cessation of the window execution. We call this *Perfect Repeatability*. Towards that end, we propose two alternative mechanisms for advancing virtual time. The first we call *Application driven virtual time* or APP-VT and the second we call *Instruction driven virtual time*, or INS-VT. Under APP-VT an application directly controls advancement of its virtual time. Just as in a discrete-event simulation the model specifies how virtual time advances as events execute, under APP-VT a modeler annotates source code to cause measurement and reporting of virtual time advance. By contrast the INS-VT approach is completely transparent to the emulated code. It more closely resembles the original TimeKeeper mechanism, except that the execution burst is defined in terms of the number of assembly language instructions the application executes, not a length of time.

In the rest of the paper, we describe both of these modes of VT advancement and highlight difficulties in their practical implementation. We evaluate overheads associated with each of them in a simple network

emulation case study using the mininet emulator, and analyze the conditions (if any) under which one or more of them may be perfectly repeatable. The paper is organized as follows: In Section 2 we introduce TimeKeeper and give a brief overview of its architecture and virtual time advancement mechanism. In Sections 3 and 4 we introduce APP-VT and INS-VT modes and describe their implementation. In Section 5, we briefly describe the experiment setup that is used in the rest of the paper to compare and contrast the three VT advancement modes. In Section 6 we use the running case study setup to measure repeatability and overheads associated with all 3 modes of VT advancement. In Sections 7 and 8, related works and conclusions are presented respectively.

2 TIMEKEEPER

TimeKeeper (Lamps et al. 2014) is a software bundle which includes a portable Linux Kernel Module and a small set of modifications to the Linux Kernel; these bring the notion of virtual time to each process placed under TimeKeeper's control. TimeKeeper manages the scheduling of these processes. Applied to network emulation, each process is typically a device (or node) in the emulated network, such as a host, switch, or router. In typical application emulation-specific scripts are used to cause the emulator to bring up all nodes and their constituent applications; these processes are all under the control of TimeKeeper. TimeKeeper processes are advanced together in a time-stepped fashion with a time-step size (in virtual time) of t. To ensure that no message sent within a time-step is also called upon to be received in the same time-step (by a process different from the sender,) a simple technique is to choose the time-step size t to be the smallest link delay in the emulated network.

TimeKeeper processes may have individual time dilation factors. To advance a process with TDF α_i by t in virtual time the original versions of TimeKeeper scheduled the process to run for $\alpha_i * t$ units of wallclock time. During each such execution burst, the dilated process may invoke system calls to query the current time (e.g. "gettimeofday".) TimeKeeper intercepts all of these system calls and returns a value based on the processes' virtual time clock. For more complex time based operations such as sleep, TimeKeeper must ensure that every such operation is enforced in virtual time. In the most recent implementation (Lamps et al. 2018) of TimeKeeper, the list of intercepted time query system calls include *gettimeofday*, *time*, and *clock_gettime*. A Linux process can also schedule sleep operations or poll on network sockets or files for new data using *sleep*, *nanosleep*, *select*, *poll*, and *timerfd*. TimeKeeper modifies the implementation of these system calls so that when called by a process under TimeKeeper administration the specified relative span is accorded in virtual time. In addition TimeKeeper can,

- Assign and dynamically change the length of the execution burst used to advance a process and automatically detect and control new processes spawned by existing dilated ones;
- Cause a stopped process to start and a running process to stop;
- Schedule TimeKeeper controlled processes so that they advance together in virtual synchrony.

With this functionality, the interface to TimeKeeper provides an emulator with the ability to advance a controlled process for a specific duration of virtual time and know when the processes have stopped at the end of a round.

For each emulated node, TimeKeeper maintains a separate scheduler queue and tracks all processes running on the emulated node. During each execution burst, round robin scheduling is imposed on processes present in a node's schedule queue which are resumed and paused using kernel level signalling techniques (SIGCONT and SIGSTOP). Thus by simply leveraging signals, TimeKeeper is able to circumvent the linux scheduler and control the order and execution durations to all processes under its control. Algorithm 1 illustrates TimeKeeper's VT advancement mechanism during each time-step. It takes as input the set of nodes to manage and a CPU mapping which specifies which CPU each node should run on. The algorithm is run in parallel on all CPUs.

Algorithm 1 *TimeKeeper_Per_TimeStep_Operation*(*CPU*)

INPUT $\{(node_i, t_i)\}$
INPUT CPU_MAP
for each node $\in$ CPU_MAP[CPU] :
  Get node's execution burst length t
  ΔClock(node) = 0 {initialize virtual time increment of node}
  for each process $\in$ node until ΔClock(node) = t :
   Compute process target virtual time advance t_p {Based on round-robin schedule among processes in each node}
   Start process execution burst for $TDF * t_p$ secs
   Handle all time operations in virtual time
   Freeze process after $TDF * t_p$ secs
   Clock(node) += t_p units, ΔClock(node) += t_p units {Advance virtual time of node by t_p units}

2.1 Remarks on Timer Accuracy

The repeatability of an experiment depends on the extent to which the emulation platform is able to reproduce each process' execution behaviour and inter process interactions at precise moments in virtual time. Without TimeKeeper the operating system controls an emulator's processes by allocating them execution quanta, and by preempting select processes in favour of others. This makes an emulation non-repeatable. With TimeKeeper, we are able to control the order and length of execution bursts to each process and thus circumvent some of the variances introduced by the Linux scheduler. But control over each execution burst is still enforced through the use of OS level timers, which still admits for timing inaccuracies. We can instruct the OS to fire a timer after s microseconds and send a signal, but actual recognition of the timer's firing is impacted by the state of the OS at the time of the firing, likewise the recognition of a signal is state-dependent. We do not have precise control over where precisely in the application code a process is when it is suspended.

We later present empirical results which show that while statistics about application behavior have considerably less variance under TimeKeeper than when the emulator is uncontrolled, it is not zero. We next present new ways of controlling a processes' execution that (under certain circumstances) make their execution repeatable.

3 APPLICATION DRIVEN VIRTUAL TIME

In the original versions of TimeKeeper, virtual time advancement was tied to wallclock time advancement with a scaling factor, the time-dilation factor. In the course of developing TimeKeeper we encountered emulations that cried out for a different means of linking execution behavior to virtual time advancement. One example was the emulation of a Programmable Logic Controller (PLC), written in python (Babu and Nicol 2016). Most of the coded executed by that emulator is python framework code and not interpretation of PLC instructions. Another example was a software defined router (in C++) we used to emulate a hardware router. A considerable amount of the code executed was in support of routing, but was not code (or even functionality) you find in hardware routers.

Correspondingly we hit on the notion of annotating the source of emulated code with calls to a software framework, the *Application Virtual Time Manager*, or AVTM, to have the application itself report its advances in virtual time. This places the burden of modeling the advancement of virtual time on a human being. The application interface to the AVTM is simple, comprised of three calls. A type one call says "I've advanced virtual time by t"; a type two call says "I've advanced virtual time by t, and I'm about to enter an IO operation." A type three call says "I've completed an IO operation". By adding the incremental advances in virtual time that are reported to it, the AVTM computes the process' virtual time clock, which

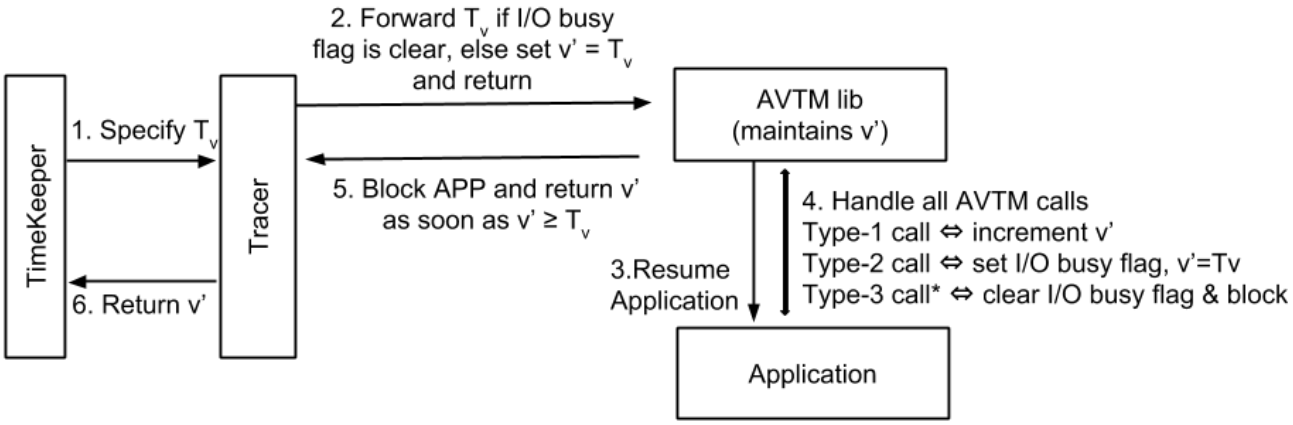

Figure 1: APP-VT architecture. The figure describes all steps involved in the interaction between TimeKeeper, Tracer and an application process in one round. It is to be noted that Type-3 AVTM calls (marked with *) are an exception and can occur asynchronously at any point in time.

is stored where TimeKeeper's Unix modifications access it when called upon to interpret a time-dependent system call.

Control of the emulated process' execution is hierarchical. TimeKeeper controls a process known as Tracer; indeed, from TimeKeeper's point of view, Tracer is the process it controls. As TimeKeeper is a kernel process and Tracer is a user process, they communicate through proc file ioctls. From TimeKeeper's point of view, it tells a process to advance up to a given virtual time T_v. Receiving this the process unblocks, does what it needs to do to advance virtual time to T_v, communicates its completion back to TimeKeeper and suspends. Receiving this TimeKeeper knows that that process has done what it needs to do in the current time-stepped round. But virtual time advance is not so clean when advanced by the application, it is possible for an application to advance through a length of time equivalent to many many time-steps all at once. Tracer is the intermediary between TimeKeeper's view of virtual time advance and what actually happens in the emulator. AVTM and Tracer maintain a Unix pipe between them. AVTM looks for direction from Tracer by performing a blocking read on this pipe. When Tracer sends a message down the pipe with a target virtual time T_v, AVTM and the emulated process unblock and the process executes. Each time the process reports a virtual time advance to AVTM the virtual clock is updated, as described below.

- If the update v' is the result of a type one call and $v' < T_v$ then the AVTM just returns control to the caller.
- If the update v' is the result of a type one call and $T_v \leq v'$, the AVTM notifies Tracer of virtual time v' through the pipe, and suspends via a blocking read on the pipe. On receiving this message, Tracer notifies TimeKeeper that it has completed the time-step.
- If the update v' is the result of a type two call, AVTM sets the virtual time to $\max\{v', T_v\}$, sets a shared flag 'IO-busy', sends the new virtual time to Tracer across the AVTM pipe, and continues with the IO. Tracer sees a virtual time at least as large as T_v and notifies TimeKeeper that it has completed the time-step. Note that AVTM and the emulated process are still executing, not under control of the Tracer. When the IO operation eventually completes the process notifies AVTM with a type three call. In response the AVTM clears the 'IO-busy' flag, and blocks on the AVTM pipe. The process' virtual time at the point of the IO completion and subsequent type three call to AVTM defines the time of the IO completion.

The special case of active IO calls for a more specific description of the interaction between TimeKeeper and Tracer. When TimeKeeper notifies Tracer to advance to virtual time T_v, the following occurs.

- If the virtual time maintained by Tracer is already as large as T_v, Tracer immediately reports to TimeKeeper that the time-step is complete, and blocks.
- If the 'IO-busy' flag is set, Tracer advances the virtual time to T_v, reports to TimeKeeper that the time-step is complete, and blocks.
- Otherwise Tracer sends a message over the AVTM pipe for the process to advance at least as far as T_v.

The key point is that suspension is tied to a specific line of source code, so that every re-animation of the process is a return from that call. This gives greater control over repeatability than did the timer-based control of execution bursts. That said, it is still possible for a APP-VT controlled emulation to not be perfectly repeatable. The source of non-repeatability is IO. The solution we describe allows the advancement of virtual time on a process executing IO to depend on that IO's physical timing. It is conceivable that by directly modeling the duration of IO operations we could ensure perfect repeatability, and will examine that approach in future work. Figure 1 is a compact illustration of the APP-VT architecture and all of the above described steps.

4 INSTRUCTION DRIVEN VIRTUAL TIME

In the previous section, we described APP-VT. It is clear that in APP-VT, the virtual time advancement is not transparent to the application's source code, which places a modeling burden on the user, and in any case is only possible when the source code of the application being emulated is accessible and can be modified and re-compiled. The question we then try to answer is: Can we design a mechanism which is transparent to the application's source code while maintaining perfect repeatability whenever possible ?

The instruction driven virtual time mechanism is an alternative approach that can satisfy both requirements under certain conditions. It is based on the idea of specifying every execution burst in terms of number of instructions to execute instead of a time duration. The execution burst duration must be converted to a specific number of instructions and we assume that the user provides this conversion ratio at the start of the emulation. The INS-VT architecture is similar to the APP-VT setup where TimeKeeper interacts with a tracer process. Here a tracer may control multiple applications. When TimeKeeper schedules a process to run, it notifies the tracer of the target upper edge of the virtual time window. The tracer converts this into a number of instructions to run, and then does so. In our practical implementation, we use the **ptrace** interface available in the Linux kernel to control execution behaviour at instruction level granularity. The ptrace interface is a fundamental component of many popular debuggers like gdb and it can be used to single step instructions by setting a trap flag on the x86 processor. It treats each system call as one user instruction and single steps over them as well. Using the ptrace interface, the tracer process can simply initiate a single step of the application and wait until the single step completes before repeating it again for the specified number of instructions. The pseudocode of the tracer process is illustrated below in Algorithm 2.

One of the subtle challenges that comes with using the ptrace interface is dealing with system calls that may block. On single stepping a blocking system call (e.g sleep), the single step operation would complete only after the system call returns and the tracer process would be blocked in the mean time. This would not bode well for the virtual time advancement mechanism because the current round can only be completed when all applications run their assigned number of instructions. To prevent such a situation, we implement kernel level modifications to the linux scheduler (in particular to the $schedule()$ function in the linux kernel) and detect if a controlled application process voluntarily relinquishes the CPU inside a system call. Upon detection of a voluntary CPU yield, the tracer process is notified and a "block flag" is set for the application. The block flag is only cleared upon exit from the blocked system call.

One of the advantages of INS-VT over APP-VT is the independence of the virtual time advancement mechanism from the application's source code. In addition, it can achieve perfect repeatability unless the control flow is altered on rare occasions by I/O device interaction failures (e.g busy disks) which may

Algorithm 2 *Tracer_Operation*()

```
while Experiment not stopped do
  INPUT {(APP_i, Ninstructions_i)} {At start of round, input number of instructions to execute}
  for each APP_i in Tracer's control do :
    for j = 0; j < Ninstructions_i; j++ do
      if APP_i is blocked then
        break
      else
        Initiate single step for APP_i
        Wait until single step is complete or APP_i is blocked
      end if
    end for
  end for
end while
```

cause certain system calls to sometimes block or fail. But as we observe in our evaluations, INS-VT in general performs very well in reproducing experimental results. On the downside, it suffers from increased overheads because each atomic single step operation generates a trap and forces execution of hundreds of additional instructions. These overheads increase the overall experiment completion times significantly. Furthermore, aligning virtual time advancement with the number of instructions executed make sense only with compiled code, and not interpreted code. APP-VT can separate code execution that advances virtual time from code execution that does not in a way that neither original TimeKeeper nor INS-VT can achieve.

5 EXPERIMENT SETUP

In this section, we briefly describe an example that is used in the rest of the paper to compare and contrast the three virtual time advancement mechanisms. The emulated network is configured as a binary tree with 3 levels. The four leaves are hosts while the remaining three nodes are switches. Hosts and switches are numbered increasingly from left to right as depicted in the Figure 2. Host pairs $h1-h3$ and $h2-h4$ communicate with each other with $h1$ and $h2$ acting as clients. Clients repeatedly send requests and wait for responses from servers before immediately sending the next request. Custom C implementations of hosts and switches are used and the topology is emulated with the mininet emulator on a standard Dell Laptop with 8 cores and 16GB of RAM. All links were assigned a delay of $1ms$. We estimated virtual time advancement values used in APP_VT by attempting to measure them. We used a fixed per-round virtual time window size of $100\mu s$ (which is smaller than all link delays) in all experiments. In our INS_VT experiments, we used a conversion ratio of 1000 instructions per μs which translates each execution burst to 100,000 instructions.

6 EVALUATION

Table 1: Maximum RMS errors between each of the 5 sets of experiment runs for every configuration. The RMS errors are computed (in secs) on the delays experienced by the first 1000 packets at $h1$.

Normal	TimeKeeper-VT	APP-VT	INS-VT
3.845e-04	1.0126e-04	0.0	4.3817e-05

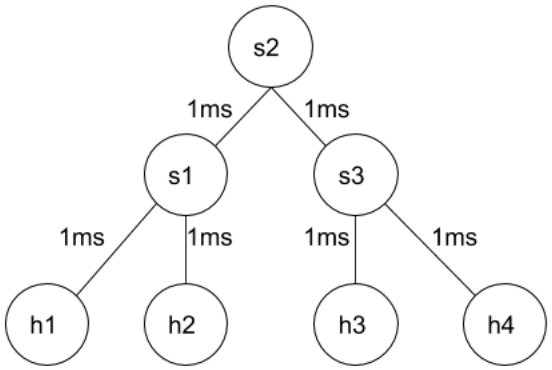

Figure 2: The simple binary tree topology used in experiments. It was emulated with mininet. Nodes $h1$, $h2$ run a client application while $h3$ and $h4$ run a server application. $h1$ sent requests to $h3$ and $h2$ requested from $h4$.

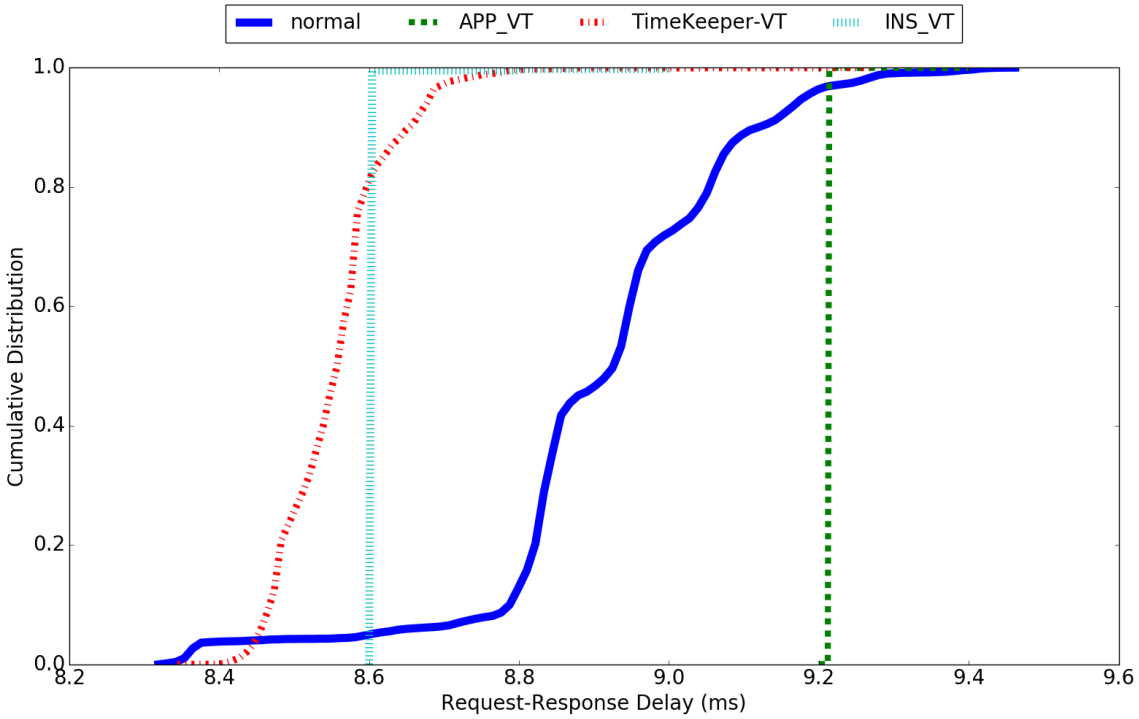

Figure 3: CDFs of round trip times experienced by all packets at $h1$ and $h2$ across the 5 experiment runs for each configuration.

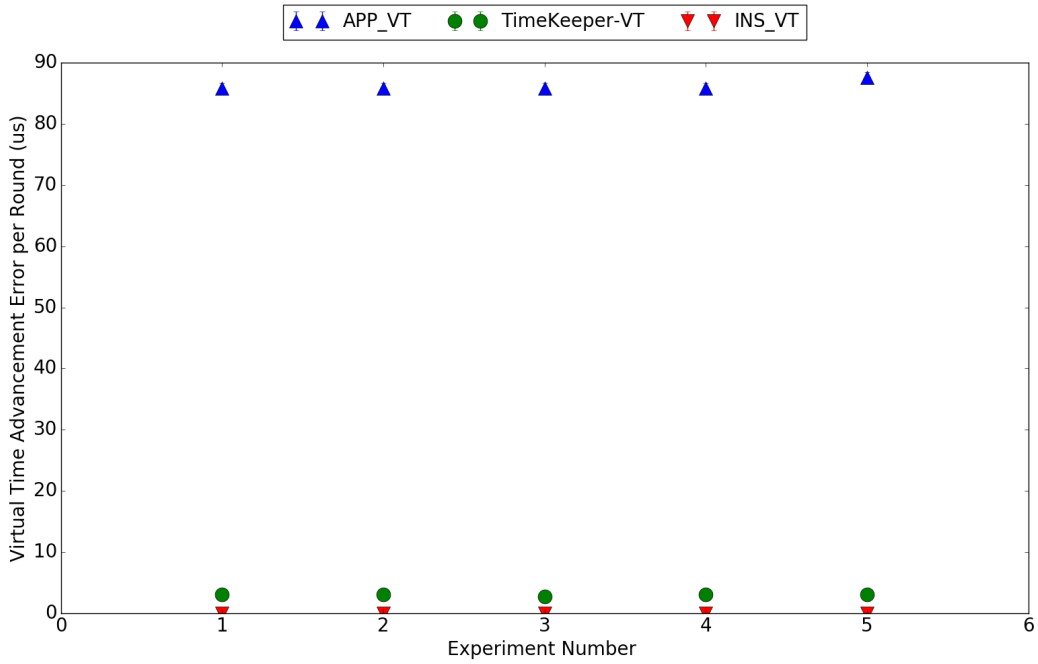

Figure 4: Mean and standard deviation of round overshoot in each configuration.

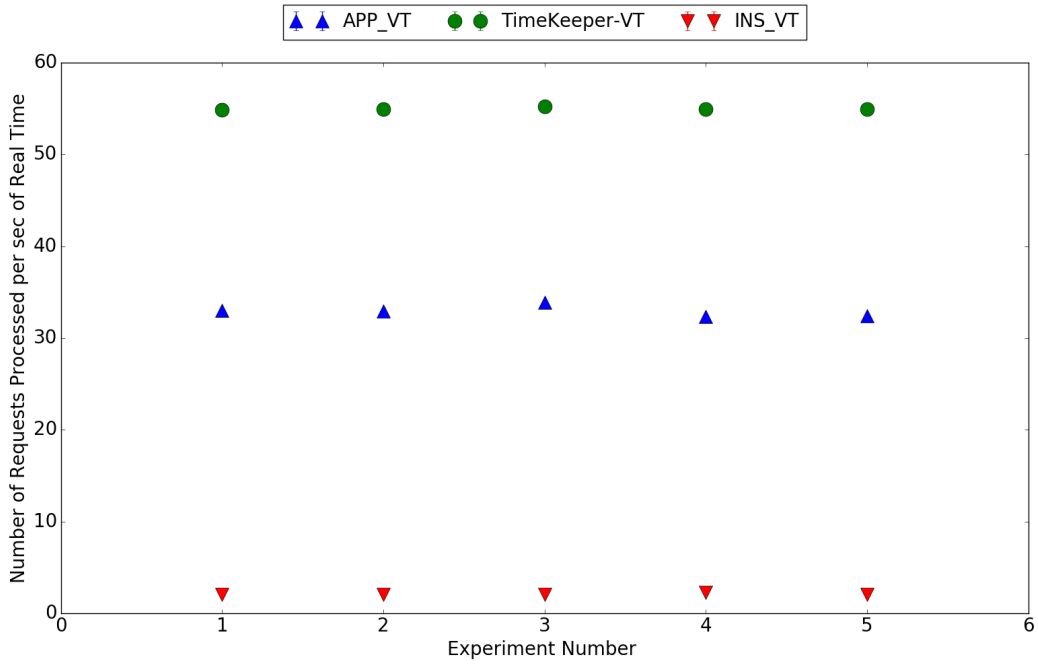

Figure 5: Measure of overhead: number of requests served per sec of real time.

In this section, we empirically measure repeatability and overheads associated with each of the three modes of VT advancement. We use the setup described in the previous section and look at 4 different configurations: (1) normal: without any virtual time, (2) TimeKeeper-VT with TDF 1 (3) APP-VT and (4) INS-VT. We repeated the experiment 5 times for each configuration.

6.1 Measuring Repeatability

During each experiment run, the two clients send requests to their respective servers and measure the delay between the send time of each request and the receive time of its response. Each client sends the next request as soon as the response to its previous request is received and processed.

For each experiment run in a specific configuration, we look at the round trip delays experienced by the first 1000 packets received at $h1$ and treat it as a vector. For each configuration, we compute the maximum RMS error between all pairs of corresponding delay vectors. Table 1 contains the measured maximum RMS error values for each configuration. A lower max RMS error value indicates that even across experiment runs, packets more or less experienced similar round-trip times. From the table it is clear that without virtual time scheduling the RMS value is higher than any other scheduling policy. When operated in APP-VT mode, round trip times observed across multiple runs appear perfectly identical to each other. We observe non zero RMS errors for TimeKeeper-VT and INS-VT which suggests some degree of variability. But RMS errors could potentially be swayed even by 1 or 2 packets experiencing abnormal delays and thus we need a more detailed picture to reason about these errors. To draw deeper insights, we plot the CDFs of delays experienced by all packets at $h1$ and $h2$ across all 5 runs for each configuration. (Figure 3).

From Figure 3, it is clear that with INS-VT and APP-VT, most packets experience near identical delays across experiment runs. That the delays are *different* is a result of the different ways of advancing virtual time. INS-VT may be counting instructions that APP-VT does not directly attribute to advancing virtual time, and/or the virtual delays APP-VT reports at the source code level don't align precisely with counts of instructions at the assembly language level.

Original TimeKeeper and uncontrolled execution show more spread out distributions. This can be explained by Figure 4 which shows the "per round overshoot" in virtual time for each mode. Since TimeKeeper cannot always precisely control the execution burst duration (due to timer inaccuracies), processes may overshoot or run a little bit extra during each round. For the original versions of TimeKeeper this caused variance in execution behaviour across rounds within the same experiment and across different experiment runs eventually leading to more spread out delay distributions. Overshoots are also common in APP-VT as illustrated in the Figure, but aren't an error, as the application state and associated virtual time are in perfect alignment.

6.2 Measuring Overhead

To measure overhead associated with each mode, we compute the total number of client-to-server requests which were completed per second of real time. A higher value indicates that the experiment is performing more useful "work" per second and thus it will take less physical time to complete a required amount of "work", or (equivalently) advance through a given length of virtual time. From Figure 5, we observe that overhead associated with INS-VT is an order of magnitude higher than others because single stepping at an instruction level is quite expensive. With TimeKeeper's mechanism, overhead is low because scheduling timers to control execution bursts is less expensive. The overhead associated with APP-VT is higher than TimeKeeper because of the multi-step interaction between the TimeKeeper, Tracer and the application process (Figure 1).

7 RELATED WORK

The concept of embedding virtual time in emulation is not new. In (Gupta et al. 2005), time dilation was put forward as a potential solution to improve scalability of hybrid emulation-simulation systems. Time dilation factor (TDF) was defined as the ratio of rate of progress of real time to virtual time. SVEET! (Erazo et al. 2009) is a TCP protocol evaluation testbed built using the proposed time dilation technique. It stresses on the importance of virtual time systems in performance analysis of emerging technologies and demonstrates the cost benefits of time dilation by accurately predicting TCP performance on slower

hardware. In (Zheng et al. 2012), the authors adopt a new approach to advancing virtual time which unlike Gupta's solution, is less constrained by the advancement of real time. The proposed virtual time system includes a virtual time control phase to decide how far each container should advance in virtual time. Virtual synchrony is maintained by blocking containers which have advanced too far in virtual time to allow others to catch up. This work is extended in (Jin and Nicol 2015) specifically to simulation of software defined networks (using OpenVZ) and in (Yan and Jin 2017) using Linux containers. TimeKeeper (Lamps et al. 2014) builds on them by bringing the notion of TDF to the forefront and allows more finer control over process execution times with small modifications to the Linux Kernel.

8 CONCLUSION

Virtual testbeds are a critical tool needed to help assess the cyber-security (and/or assess the effectiveness of cyber-security protection mechanisms) of embedded systems such as those that control the nation's electric power grid. A desirable attribute of a virtual testbed is an ability to be able to precisely repeat behavior, given the same initial state and exactly the same sequence of events, or inputs. Such an ability would help analysts to understand precisely the sequence of events leading to a vulnerability exploit, to understand how any other unexpected behavior came to pass, or even to debug the software and models comprising the virtual testbed.

However, it is difficult to guarantee repeatable emulations because the OS and underlying hardware have behaviors the emulation does not control, which leads to variance that can affect reproducibility of timings associated with inter process interactions. We saw that by changing how virtual time is advanced and by controlling their execution order and duration, we can reduce some OS level interference effects and improve repeatability. We illustrate these improvements through empirical evaluations with an open source virtual time system for Linux called TimeKeeper. However, we noted that TimeKeeper's reliance on OS level timers to control execution bursts can still admit non-reproducible variances in execution behaviour. We proposed and analyzed two alternative modes of virtual time advancement (1) APP-VT: the application controls advancement of its own virtual clock and (2) INS-VT: each applications virtual clock is advanced based on the number of instructions executed. We empirically demonstrated that with both APP-VT and INS-VT one could achieve near perfect repeatability. APP-VT places the burden of defining virtual time advance on the modeller but can be used with emulators where the measuring execution time or counting instructions is not a good measure of virtual time advance. By contrast INS-VT is completely transparent to the modeler. Of the three, the technique of measuring execution bursts has the least overhead but the greatest demonstrated variability in behavior. INS-VT has the least demonstrated variability in behavior, but the highest overhead. APP-VT is less than a factor of two times slower than original TimeKeeper, has significantly better repeatability, but requires significantly more work on the part of the modeler to advance virtual time than either original TimeKeeper or INS-VT. The modeller is thus ultimately left with a choice to trade-off overhead with ease of implementation. In future work, we would explore the possibility of creating simple and easy to use APP-VT interfaces and reduce overheads associated with INS-VT.

ACKNOWLEDGEMENT

This work was supported in part by the Siebel Energy Institute, in part by the Boeing Corporation, and in part by Department of Energy under Award Number DE-OE0000780. Disclaimer: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and

opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

REFERENCES

Ahrenholz, J., C. Danilov, T. R. Henderson, and J. H. Kim. 2008. "CORE: A Real-time Network Emulator". In *Military Communications Conference, 2008. MILCOM 2008.*, 1–7. IEEE.

Babu, V., and D. M. Nicol. 2016. "Emulation/Simulation of PLC networks with the S3F network simulator". In *In Proceedings of the 2016 Winter Simulation Conference*, edited by T. M. K. Roeder et al., 1475–1486. Piscataway, New Jersey: IEEE.

Erazo, M. A., Y. Li, and J. Liu. 2009. "SVEET! a Scalable Virtualized Evaluation Environment for TCP". In *Testbeds and Research Infrastructures for the Development of Networks & Communities and Workshops, TridentCom 2009.*, 1–10. IEEE.

Gupta, D., K. Yocum, M. McNett, A. C. Snoeren, A. Vahdat, and G. M. Voelker. 2005. "To Infinity and Beyond: Time Warped Network Emulation". In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, 1–2. Brighton, United Kingdom: ACM.

Jin, D., and D. M. Nicol. 2015, December. "Parallel Simulation and Virtual-Machine-Based Emulation of Software-Defined Networks". *ACM Transactions on Modelling and Computer Simulation* 26(1):8:1–8:27.

Lamps, J., D. M. Nicol, and M. Caesar. 2014. "TimeKeeper: A lightweight Virtual Time System for Linux". In *Proceedings of the 2nd ACM SIGSIM/PADS conference on Principles of Advanced Discrete Simulation*, 179–186. Denver, Colarado: ACM.

Lamps, J., V. Babu, D. M. Nicol, V. Adam, and R. Kumar. 2018. "Temporal Integration of Emulation and Network Simulators on Linux Multiprocessors". *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 28(1):1–25.

NRL 2010. "Emane - Extendable Mobile AdHoc Network Emulator". http://www.nrl.navy.mil/itd/ncs/products/emane.

ONF 2016. "Mininet- An Instant Virtual Network on your Laptop". http://mininet.org/.

Yan, J., and D. Jin. 2017. "A lightweight Container-based Virtual Time System for Software Defined Network Emulation". *Journal of Simulation* 11(3):253–266.

Zheng, Y., D. Nicol, D. Jin, and N. Tanaka. 2012. "A Virtual Time System for Virtualization-based Network Emulations and Simulations". *Journal of Simulation* 6(3):205–213.

AUTHOR BIOGRAPHIES

VIGNESH BABU is currently a Graduate student and Research Assistant in Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign. His research is primarily focused on modelling and analysis of cyber security issues in the Smart Grid. His email address is babu3@illinois.edu.

DAVID M. NICOL is the Franklin W. Woeltge Professor of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign, and Director of the Information Trust Institute. He is the PI for two national centers for infrastructure resilience: the DHS-funded Critical Infrastructure Reliance Institute, and the DoE funded Cyber Resilient Energy Delivery Consortium. His research interests include trust analysis of networks and software, analytic modeling, and parallelized discrete-event simulation,research which has lead to the founding of startup company Network Perception, and election as Fellow of the IEEE and Fellow of the ACM. He is the inaugural recipient of the ACM SIGSIM Outstanding Contributions award. He received the M.S. (1983) and Ph.D. (1985) degrees in computer science from the University of Virginia, and the B.A. degree in mathematics (1979) from Carleton College. His email address is dmnicol@illinois.edu.