Fast Switch into a Trustworthy Virtual Machine for Running Security-Sensitive Applications

Adrian Pop
Computer Science Department
Technical University of Cluj-Napoca

Adrian Coleșa
Computer Science Department
Technical University of Cluj-Napoca

Andrei Luțaș
Computer Science Department
Technical University of Cluj-Napoca

Alexandru Gurzou
Computer Science Department
Technical University of Cluj-Napoca

Sandor Lukacs
R&D Department
Bitdefender

Raul Toșa
R&D Department
Bitdefender

Daniel Țicle
R&D Department
Bitdefender

***Abstract*—We aim to protect an end-user security-sensitive application by running it apart from the untrusted applications in a trusted virtual machine (VM), named the green VM. The untrusted applications are run in another, untrusted red VM. The two VMs are isolated by one another by a small, trusted hypervisor (HV). The user must switch from the red VM into the green VM when he wants to run the protected application. Therefore, this kind of switch should be fast enough for usability reasons. We propose different methods to build an as small as possible green VM, prepared to run only the protected application, though above a full-featured operating system (OS), in order to avoid rebuilding the application and support various hardware on the client system. The resulting VM would be more trustworthy, so more secure, and also could be booted faster, thus reducing considerably the VM-switch time.**

I. Introduction

Virtualization was heavily used in the recent years to provide a better security [1] for computing systems. One of the main advantages the virtualization brings is the isolation [2] of the security tools from the protected system. The security tools could be placed in the virtualization OS, i.e. the hypervisor (HV), while the protected system runs inside a virtual machine (VM) above the HV. This way the security tools run at a higher privilege level than the protected system, which results in a better isolation from that system and a better control of it.

The traditional virtualization-based security strategy is based on analyzing the protected VM's state from the HV and reacting to malicious actions or contents. Such a strategy, named *introspection* [3], was used for both intrusion detection and prevention. The main problem the virtualization-based introspection faces is the *semantic gap* [2], which makes semantically understanding the VM's memory contents difficult. There are different ways to narrow the semantic gap, based on two strategies [4]: (1) out-of-bound delivery of semantic information, a strategy that is highly dependent on and specific to the guest OS in the introspected VM [5], however providing good precision and fine-grained security-relevant details;

(2) derive VM's semantic information through semantic knowledge of the hardware architecture [6], [7], a strategy that is guest OS's agnostic, however providing low precision and only course-grained details.

Another proposed strategy was to run the applications in two different VMs [8], [9], based on their sensitivity to security aspects: (1) important, security-sensitive applications in a trusted VM, named the *green VM*, and (2) the other applications in a different, untrusted VM, named the *red VM*. In the red VM the user has the freedom to run any kind of applications, perform any kind of operations and interact with any external parties (e.g Web sites) he wants, but at the expense of weaker security guaranties. On the other hand, in the green VM the user is highly restricted regarding what applications he can run (usually only the protected security-sensitive ones) and which external parties he can interact with (usually only the sites the protected applications trust). The restrictions imposed in the green VM make it more trustworthy. Another way to improve the green VM's trustworthiness is by reducing its size as much as possible, based on the principle that smaller is simpler and easier to verify, so more reliable and trustworthy.

Our paper is based on a variant of the red-green VM protection strategy, which keeps active only one VM at a time [10], [11], with direct access to the I/O devices, while having the other VM shutdown. This is different from the alternative that keeps both VMs active simultaneously and virtualizes the I/O devices. The former strategy requires a much smaller and less complex HV, having no need to virtualize the I/O devices.

The price for the security benefits of our strategy is the switch time between the two VMs, so we tried reducing it as much as possible, in particular, reducing the switch time from the red VM to the green VM. One important factor in this context is the green VM's startup time, which depends on the VM's image size. Therefore, we propose some methods to reduce the green VM's size. In order to simplify our explanations we consider in the following the case of a single protected application run in the green VM. Getting a smaller green VM also accounts for a reduced attack surface of that VM, so a better protection of the application run in it.

The main idea behind getting a small green VM is to build its software stack specific to the only application that will be run inside it, i.e. the application itself, the libraries and OS components it uses. There are basically two strategies that try doing that by building the VM as a (1) a *container* [12] on the same OS shared with other containers, or (2) a *light VM* [13], [11], [14], customized for the single application run in it, having its own OS, different from the other VMs.

The container-based strategy saves space due to sharing the same OS between multiple containers. It also provides a short instantiation and startup time for a new container. The price for those features is however the weaker isolation between containers and thus the weaker security they provide.

Running the application to be protected in a VM better isolates it from the other untrusted applications, but requires having multiple OSes in multiple VMs. Also, the instantiation and startup times are much higher than for containers. It was recently shown however [14] that it is possible to build a really light VM getting even better performance than containers, but with a much stronger isolation and security. They are based on linking together in a single address space the application itself and, as libraries, the OS components it depends on. The resulting system is called a *unikernel* [15].

This is why we chose to isolate our protected security-sensitive application in a real VM, while reducing that VM's size to fit its minimal needs. However, our strategy is different by the unikernel one, by creating a sort of traditional VM, with two separate spaces — user and kernel, yet trying to reduce both of them to the minimal needs of the single user application run in that VM. This way, even if the resulting VM image is much larger than a unikernel's one, though much better than that of a general purpose VM, we provide better compatibility with the existing applications, not being limited by the restrictions imposed by the use of a library OS.

The main contributions of our paper are:

- a method to reduce the software stack of the green VM, still maintaining the compatibility with existing applications and also supporting naturally (as the used OS supports) all sort of hardware the client could have on his system;
- a method to switch faster to the green VM, taking advantage of the small size of the green VM's image;
- a security analysis of the proposed methods.

The following paper's structure is: Section II briefly describes the protection strategy based on running the security-sensitive application in a different, trusted, green VM and also the threat model we consider, Section III presents the method we propose to automatize the building of a small green VM, customized on the minimal needs of the single application run inside it, Section IV presents different methods we use to reduce the switch time to the green VM, Section V is about the way we evaluate the functionality of our proposed methods, Section VI compares our method with other similar projects, and Section VII concludes this paper.

II. Red-Green VM-Based Protection

A. Threat Model

The attacker could take complete control of the red VM, trying to further compromise the HV and the green VM, although we consider the latter two trusted, so together with the hardware composing our trusted computing base (TCB).

We do not consider DoS attacks as the attacker could shutdown the red VM or block the user attempts to switch from red VM into the green VM in ways that cannot be easily told apart by similar regular user actions. However, assuming the attacker's target would be the protected security-sensitive application in the green VM, DoS attacks actually also stops the attacker himself from reaching its purpose. Phishing attacks, even if very possible and important, are also out of scope of this paper.

In case the protected application represents the client-side of a client-server application, we consider the communication channel between the client and server untrusted, although the server trusted.

In order to be able to implement our protection strategy the minimal assumptions we make is that the user's system provides support for hardware-based virtualization. We also consider only one security-sensitive application to be run inside the green VM, like for instance a graphical Web browser accessing an online banking application. The problems related to dealing with multiple independent security-sensitive applications on the same user system are out of scope of this paper.

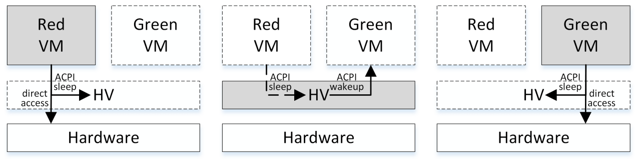
B. Strategy and Architecture

The protection strategy based on red-green VMs works by isolating the security-sensitive application (which we want to protect) from the other applications and the OS they run on, i.e. the untrusted environment, and run it in a trusted environment. Each environment is hosted by a different VM: the trusted one by the green VM, the untrusted one by the red VM.

The *trustworthiness of the green VM* is given by several factors [11]: (1) its reduced size and contents, fitting the minimal needs of the protected application, thus exposing a reduced attack surface, (2) user's restriction (imposed by HV) to run only the supported application and only with the minimal privileges needed, (3) denial remote access to the user system (i.e. all ports closed), (4) application's restricted remote access only to trusted sites, (5) application's restricted access to the local file system, and (6) the volatile nature of the green VM's status, i.e. not saving its state when suspended, and supporting only an in-memory file system.

In order to reduce the complexity of the HV, only one VM is running at one moment, while the other one is shutdown or suspended. The active VM is given direct access to the I/O devices, although the critical memory the HV and its data reside in is protected using IOMMU virtualization (e.g. Intel VT-d). This not only results in an improved I/O performance and reduced complexity of the HV (not needing to virtualize the I/O devices), but also provides an easy way to ensure trusted I/O channels between the user and the green VM.

The HV controls the VM-switch using the ACPI mechanism: 1) injects an ACPI sleep event in the active VM, forcing its OS to save that VM's state and try powering down the VM, 2) intercepts the ACPI shutdown command and does not let it affect the physical machine, 3) injects an ACPI wake-up



(a) Untrusted phase (b) VM switch (c) Trusted phase

Fig. 1: Alternative execution of the red and green VMs and the way the VM-switching is performed. The gray boxes indicate the running software component

event in the VM that must be switched to, forcing its OS to restore its state and resume its execution. Figure 1 illustrates the VM-switching mechanism. This strategy releases the HV from saving and restoring any VM's state and also from maintaining any state of the I/O devices, which are supposed to be reinitialized any time a VM wakes up and resumes its execution. Getting the I/O devices in a fresh state any time a VM-switch is performed also blocks any attack attempts from the red VM to the green VM using the I/O devices.

Controlling the VM-switch could need HV-specific modules to be run inside the guest OS of each VM. Such a module in the red VM should be protected by the HV, an operation which could be very OS specific, but this would be needed only to reduce the range of DoS attacks. There would be otherwise no way for an attacker to use it to compromise the green VM.

III. Green VM's Structure and Construction

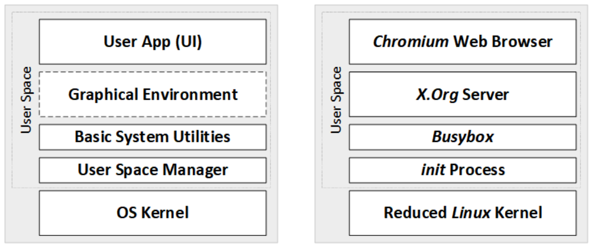
We aimed for building a green VM having an attack surface as small as possible, trustworthy for being used to run inside it the security-sensitive application we want to protect.

This is based on two main mechanisms:

1) *include in the software stack* of the green VM (both the kernel and user space) the *minimum needed* to run only the security-sensitive application we want to protect, and
2) provide a *restricted running environment* to the protected application, which cannot be escaped from.

This section describes how we implemented the two mechanisms for our particular green VM used to protect an online Web application and the ways our method could be generalized to automatically apply it for building a green VM for any other particular user application.

Figure 2a illustrates the general structure of the software stack the green VM should have. The "*User App*" is the only user application we run in the green VM, and the only user-interface (UI) with that VM. The contents of the "*Basic System Utilities*" is dependent on the supported user application. The "*Graphical Environment*" is optional, in case the supported application needs it (as was the case of our particular green VM illustrated in Figure 2b). We will detail below what each component should include and the way it could be built. Even if our method could theoretically be applied to any OS and applications it supports, our description focuses on Linux, as being the most common open-source OS at hand.



(a) For a general user application (b) For a graphical Web browser

Fig. 2: Green VM's software stack

A. Building the Linux Kernel

1) Configuration: We intended to build a small kernel, thus the configuration options were trimmed down as much as possible. There are however two perspectives we considered regarding this problem:

1) the top-down or the *user perspective*, based on which the kernel should contain only the components needed by the user application run in the green VM, and
2) the bottom-up or the *hardware perspective*, from which our kernel and green VM should be able to run on various hardware the users could use in real-life.

There are two possible starting points:

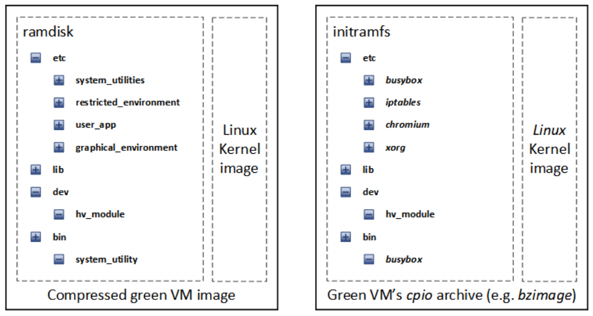
1) We can begin with running "`make tinyconfig`" (on versions which support this target), which will produce a minimal kernel configuration. What follows is *turning on* other required options.
2) The second way is to issue "`make defconfig`" and disable superfluous options. The `nm` program can be used to sort the kernel symbols by size (i.e. by running "`nm -size-sort vmlinux`"). This can aid us in finding the most relevant options that can be *turned off*, but requires the kernel to be built before, because `nm` takes the kernel image as argument.

We adopted the latter workflow because the *tinyconfig* target has been added only relatively recently to the kernel. Furthermore, as our solution is meant to support a variety of hardware, starting with a default set of drivers speeds up the configuration process. Graphical support could be enabled in the kernel configuration to allow the user-space to run a potentially graphical user application.

Once our configuration was finalized, little to no changes were required to support different applications.

2) Initramfs: The green VM being non-persistent, a simple initial memory-based disk, i.e. *ramdisk*, can be used for the root file system. We used the *initramfs*, which is a *cpio* type archive, containing a basic user-space setup. It is extracted and mounted into the root file system during the kernel boot. The kernel would execute the */init* file from the *initramfs* as the first process (PID 1).

The archive is created and bundled with the kernel image automatically during the build process by setting the `CONFIG_INITRAMFS_SOURCE` option to the name of the



(a) For a general user-application

(b) For a graphical Web browser

Fig. 3: The green VM's image layout: an archive containing the kernel image and the ramdisk with the user-space components

cpio list file. Such a file describes what is to be included with the *cpio initramfs* archive and has a particular syntax described by the `gen_init_cpio` utility from the Linux sources.

We created a standard Linux file system hierarchy and added the files we wished to populate the *initramfs* with. These were the required scripts, utilities, libraries, firmware files, and the configuration files from `/etc` (`group`, `passwd`, `shadow`).

We also specified the device node files which reside in `/dev`. One way to obtain the relevant `/dev` nodes to include in the *initramfs* was to search for such strings in the binary files from the application. This can be automated using the `strings` utility. A typical minimalistic set of `/dev` nodes is: `/dev/console`, `/dev/null`, `/dev/random`, `/dev/tty`, `/dev/urandom`, `/dev/zero`.

Beside the basic system configuration files in "`/dev`" and "`/etc`", other files the *initramfs* file system should contain are those needed by running all the user-space components of the supported user-application, i.e. binary and configuration files composing the "*Basic System Utilities*", the optional "*Graphical Environment*", the user application itself and maybe other utilities needed to ensure the restricted environment we want to confine our application. The way we automatically identified those files is described below in Section III-B.

3) Kernel Modules: Any custom kernel modules are copied to the `/lib/module/`uname -r`/` in the *initramfs*.

Figure 3a illustrates the structure of a general green VM's compressed image.

B. Building the User Space

Our build system consists of a series of scripts and is extensively automated, requiring running a simple `make` to create a standard image.

The user-space programs are built using simple scripts, performing five steps: (1) download, (2) preparation, (3) configuration, (4) compilation, and (5) installation.

The source code URL from where the program is downloaded has to be specified, along with the program's build dependencies. The preparation step makes needed changes, such as applying patches to the program's sources. The program is then configured, built and installed on the *initramfs*, and the library dependencies are copied.

Their requisite shared libraries are obtained using the `ldd` utility and they are copied to the *initramfs* memory disk. The binaries are divested of unnecessary data to reduce their size using the `strip` utility.

As opposed to using a full-fledged package manager, this system can ensure that the smallest possible version of the application is built. On the other hand, if a large number of application were to be used, using or adapting an existing package manager would speed up the developing process. Our scenarios used only a handful of programs, so the simple scripts provided a more convenient solution.

C. Building the Restricted Environment

Mitigations have to be set in place both for outside attacks (using the network) and potentially malicious user actions.

First of all, to obviate the user's interference with the system, he was allowed to run only the dedicated application, provided with no shell or other means to interact with the green VM. The application was started under an unprivileged user in a *chroot*, being as restricted as feasible. This was made possible by first installing the application, during the VM image build process, with all its dependencies inside the chroot directory. If, for example, the application would be a shell, the same restriction would apply, all the required programs would have to be installed to that location, otherwise they wouldn't be accessible.

The runtime environment was set up from the *init* script directly, which initialized the chroot.

Secondly, to hinder network attacks, basic firewall rules were set up with *iptables*. Inbound connections were denied. The specifics would depend on the application being used, but generally the green VM could also restrict the access to just the servers the application needs to connect to, e.g. the Web server a Web application is hosted on.

Finally, keeping up with the security patches reduces the green VM's attack surface.

D. Debbuging Environment

For debugging purposes during the development phase of our protection system we used a particular structure of the green VM consisting in the following:

- `busybox`: a reduced shell, containing common user-space utilities like, `ls`, `ps` etc.
- `lspci`: provides information on the available PCI devices.
- `dropbear`: a small SSH server which provides a remote shell to the green VM to be able to remotely configure and inspect it;
- *no user application* by default, while if needed, the desired user application could be copied from a remote server over the network, e.g. using `scp`, in order to avoid rebuilding the entire booting image due to different testing changes of the user application; this strategy could be extended to other user-space components the

user application is dependent on and could be frequently changed during the development phase.

IV. Green VM Management and Functionality

A. Green VM's Lifecycle

The HV suspends the red VM and transfers the control to the green VM. After booting, the kernel loads a temporary root file system, *initramfs*, and executes the *init* program therein. The *init* is a shell script which initializes a minimal environment (e.g. *busybox*), takes care of networking and performs other menial setup work. The user application is than started automatically, without giving the user other interface with the system. The *init* process is configured to terminate when the single running user application is closed. When *init* is done, the HV-specific modules in green VM informs the HV that the control should be switched back to the red VM.

In a debugging context the *init* script doesn't launch the user application directly, but the user is given a shell that can be used to explicitly start the user application, possibly by firstly copying it from a remote site.

B. Reducing the Red to Green VM Switch Time

One important factor influencing the red to green VM switch time is the green VM's startup phase. This one is mostly dependent by the size of the VM and the ramfs decompression time. The latter is managed in part by choosing the fastest compression method available (e.g. LZO or LZ4), but this also results in a larger image file, while normally faster compression / decompression algorithms have a lower compression ratio. Having as few components in the VM image as possible decreases its size and improves the boot-up speed. We must be aware however that our image size cannot be on a par with the ones that are commonly used for embedded systems or small server VMs since we want to provide running support for complex user applications, such as a full-blown graphical Web browser.

Another method we used to reduce the switch time was to load the green VM's image into RAM during the HV's startup, eliminating the need to read that image from the disk each time a switch to the green VM is performed.

We also configured the green VM to have a volatile state by: (1) not saving the state of the green VM when switching back to the red VM and thus starting the green VM from the same pristine state each time a switch to it is performed, and (2) not supporting permanently saving data, using only a ramdisk. These mechanisms also improve the performance and trustworthiness of the green VM.

Figure 4 illustrates the structure of the host system's memory, comprising the HV and the two VMs, the way the green VM is started and its ramdisk. The HV is the first software loaded and run, such that it can isolate and protect all memory domains from each other.

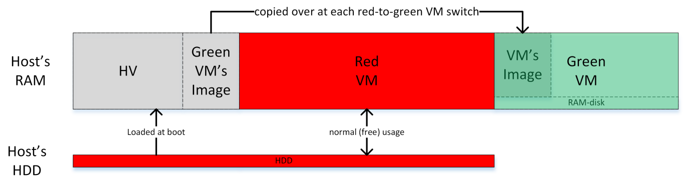


Fig. 4: Memory structure of the user's host system

V. Evaluation

A. Green VM's Software Stack for a Web Browser

We applied our method for building a light green VM used for supporting an Internet browser providing access to a remote online Web application (e.g. e-banking, e-commerce). Our VM was composed by a `busybox` (version 1.25.0) based system, together with a `Chromium` Web browser (version 31.0.1650.57) running on an `X.Org` display server (version X11R7.6) and an underlying Linux kernel. The firewall solution was provided by the `iptables` program (version 1.6.0). A full-fledged graphical Web browser, although decidedly space consuming, was required to ensure compatibility with complex Web applications. Figure 2b illustrates our green VM's software stack, while Figure 3a illustrates the structure of the corresponding *cpio* archive generated, both of them particular cases of the general proposed ones.

B. Benchmark Results

To evaluate our VM-building method both functionally and qualitatively, we applied it on three different user applications, appropriate for different usage scenarios: (1) a *graphical Web browser* for complex Web applications, (2) a *text Web browser* for simple Web applications, and (3) a *shell* we used for debugging purposes during the our method development.

Table I compares the three scenarios, contrasting them with a normal OS (an *Arch Linux* distribution), having a Web browser installed. Columns 2 and 3 represent the uncompressed user-space application sizes, together with the libraries they depend on. For the text Web browser scenario in row 3 we have used the *Elinks* browser.

The graphical Web browser setup was the most versatile, allowing interaction with most Web applications, although it resulted in the biggest VM's size, even if we configured each of the programs to be as light as possible. Therefore, only the bare minimum X.Org modules were built to support the graphical Web browser. Even so, the package takes up $81MB$. The Chromium options were trimmed down as well, resulting in a $225MB$ browser (the sizes include all required libraries).

On the other hand, the text Web browser not requiring the graphical setup, decreased the green VM image and reduced correspondingly the startup time of the green VM.

Finally, the debug image provided a full shell and a suite of debugging programs that were presented in Section III-D. We remotely connected it to copy from the network various testing versions of our Web browser and check their functionality, a strategy which was faster than building the entire VM repeatedly for each tested browser version.

VI. Related Work

There is some work [10], [11], [16] dealing with the red-green VM protection strategy, like our solution is also based

TABLE I: VM sizes and startup times without the HV

	Kernel Size [MB]	User-space [MB]		Total Image Size [MB]	Boot Time [s]
		App.	X.Org		
Default OS	92	237	95	>1000	16
Graphical Web Browser	8	225	81	136	8
Text Web Browser	8	11	-	16	4
Debug Environment	8	20	-	17	4

on, though our focus in this paper was on trying to improve the switch time to green VM, especially by decreasing as much as possible its size. In [11], [16] there are also proposed similar solutions to reduce the switch time, but we proposed here an automated method to build the green VM fitting the minimal needs of any possible user application run inside it.

LightVM [14] proposes a method to build VMs with a small image and memory footprint as well as simplifying the VMs management. Our strategy to build a small green VM is similar to theirs, though our purpose is not running many VMs simultaneously, but just one. Their VMs also run on a virtualized hardware, which simplifies their job and kernel size, while our green VM is supposed to run on various real hardware. While the application running on their VM is a server with no user interface, our VM usually must support a graphical user application (e.g. a Web browser), resulting in a significant VM image size difference in their favor. They also require changes to the guest OS to speedup the VM booting.

In the case of Unikernels [15] the obtained VM is really small by building together in a single address space the application with only the OS components it needs. Different from our VM, a unikernel-based one is thought for running in cloud a computing application with no user-interface. Unikernel strategy also requires specially-designed library-like OS components and possibly complex changes to port existing applications on this model. Our method wants to provide compatibility with existing applications and OSes, yet be general enough to be applied to any user application.

VII. Conclusions

We proposed a method to build a light VM providing support for running just one application. Our method reduces the VM's software stack to the minimum needed to run the supported application. It can also be applied to any kind of user application, automatically detecting all needed user and kernel components. Such a light VM could be used on an end-user system to provide a trustworthy, green environment for running security-sensitive applications, isolated by the other untrusted applications run in a different red VM. The small size of the green VM contributes to reducing the switch time from red to green VM, an important usability factor of our protection system.

We also proposed few other methods to reduce the VM-switch time, like always starting the green VM from a pristine state kept in memory inside the HV and using a ramdisk-based file system.

Our future work will focus on improvements of the ACPI-based suspend and resume VM-switching mechanism.

Acknowledgment

This work was supported by a grant of the Romanian National Authority for Scientific Research and Innovation, CNCS/CCCDI-UEFISCDI, project number PN-III-P2-2.1-PED-2016-2073, within PNCDI III.

References

[1] T. Garfinkel and A. Warfield, "What Virtualization Can Do for Security," *;login:, USENIX magazine*, vol. 32, no. 6, pp. 28–34, December 2007.

[2] P. M. Chen and B. D. Noble, "When Virtual Is Better Than Real," in *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, ser. HOTOS '01. Washington, DC, USA: IEEE Computer Society, 2001.

[3] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *In Proc. Network and Distributed Systems Security Symposium*, 2003, pp. 191–206.

[4] J. Pfoh, C. Schneider, and C. Eckert, "A formal model for virtual machine introspection," in *Proceedings of the 1st ACM workshop on Virtual machine security*, ser. VMSec '09. New York, NY, USA: ACM, 2009, pp. 1–10.

[5] A. Luțaș, S. Lukács, A. Coleșa, and D. Luțaș, "U-HIPE: Hypervisor-Based Protection of User-Mode Processes in Windows," *Journal of Computer Virology and Hacking Techniques*, pp. 1–14, 2015.

[6] S. T. Jones, A. C. Arpaci Dusseau, and R. H. Arpaci Dusseau, "VMM-based hidden process detection and identification using Lycosid," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '08. New York, NY, USA: ACM, 2008, pp. 91–100.

[7] S. Vogl and C. Eckert, "Using Hardware Performance Events for Instruction-Level Monitoring on the x86 Architecture," in *Proceedings of the 2012 European Workshop on System Security (EuroSec'12)*, 2012.

[8] B. Lampson, "Accountability and Freedom," {http://research.microsoft.com/en-us/um/people/blampson/slides/accountabilityAndFreedomAbstract.htm}, Sep. 2005.

[9] ——, "Privacy and Security: Usable Security: How to Get It," *Commun. ACM*, vol. 52, no. 11, pp. 25–27, Nov. 2009.

[10] A. Vasudevan, B. Parno, N. Qu, V. D. Gligor, and A. Perrig, "Lockdown: Towards a Safe and Practical Architecture for Security Applications on Commodity Platforms," in *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, ser. TRUST'12. Springer-Verlag, 2012, pp. 34–54.

[11] A. Coleşa, S. Lukács, V. Topan, R. Ciocaş, and A. Pop, "Efficient Provisioning of a Trustworthy Environment for Security-Sensitive Applications," in *Proceedings of the 8th International Conference on Trust and Trustworthy Computing (TRUST2015)*. Springer International Publishing, August 2015, pp. 300–309.

[12] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 275–287, Mar. 2007.

[13] A. Filyanov, J. M. McCuney, A. R. Sadeghiz, and M. Winandy, "Uni-directional trusted path: Transaction confirmation on just one device," in *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, ser. DSN '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1–12.

[14] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My VM is Lighter (and Safer) Than Your Container," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: ACM, 2017, pp. 218–233.

[15] A. Madhavapeddy and D. J. Scott, "Unikernels: Rise of the Virtual Library Operating System," *Queue*, vol. 11, no. 11, Dec. 2013.

[16] A. Coleșa, S. Lukács, V. I. Topan, and R. Ciocaș, "Server-Triggered Trusted User Confirmation of Transactions Performed on Remote Sites," *Automation, Computers, Applied Mathematics (ACAM)*, pp. 1–11, 2015.