A secure architecture design based on code minimization and application isolation

Aditi Gupta
Purdue University
West Lafayette, IN
aditi@purdue.edu

Michael S. Kirkpatrick
James Madison University
Harrisonbourg, VA
kirkpams@jmu.edu

Elisa Bertino
Purdue University
West Lafayette, IN
bertino@purdue.edu

ABSTRACT
With fast evolving attacks, using software patches for fixing software bugs is not enough as there are often considerable delays in their application to vulnerable systems and the attackers may find other vulnerabilities to exploit. A secure architecture design that provides robust protection against malware must be guided by strong security design principles. In this work, we propose a system design based on the security principles that aim at achieving isolation and reducing attack surface. Our design leverages multi-core architecture to enforce physical isolation between application processes so that a malicious or infected application is unable to affect other parts of the system. Further, we significantly reduce the software attack surface by executing each application on its own customized operating system image that is minimized to only contain code required by the given application.

1. INTRODUCTION
Recent years have seen a rise in sophisticated malware that employs various techniques, such as return oriented programming (ROP), to bypass system defenses. A typical response to such malware attacks is by patching the software vulnerability that was exploited to launch the malware. This involves deploying patches to vulnerable systems which involves considerable delay and the attack might have already propagated by the time the patch is applied. Further, patching vulnerabilities is not an ideal solution to building secure systems as attackers are constantly finding new vulnerabilities and crafting new exploits. A secure architecture design that is resilient to evolving malware must be based on strong security principles that minimize the risk and limit damage if some part of the system is compromised. Further, these solutions should not be limited to software layer security but should also enforce protection at hardware layer.

In this paper, we propose a design for secure architecture that is guided by two principles derived from the well known security design principle of least privilege. The first is *application isolation* which requires that applications be isolated from each other, thereby preventing a malicious application from compromising other applications. Often, isolation is implemented using software isolation techniques such as virtualization. It has been shown that software isolation is not sufficient since a malicious application may exploit vulnerabilities in the virtualization software to attack other parts of the system [11, 7, 6]. Also, software isolation techniques have been shown to be ineffective against side channel attacks that exploit shared channels such as memory usage statistics for accurate adversarial inference of other processes' behavior [5]. Thus, software isolation is insufficient and secure systems must incorporate physical isolation by isolating the hardware resources as well.

The second principle of our design is *code minimization* which reduces the software attack surface available to an attacker. Widely used operating systems (OS) such as Windows and Linux are based on monolithic kernels, that is, the entire OS code executes with superuser privilege. This provides a huge computing base that can be potentially exploited by the attacker to perform privilege escalation attacks. Also, a typical process may use only a small fraction of services provided by a kernel. This leaves a significant chunk of privileged code base that is unused code. Our principle of code minimization aims at minimizing the kernel code so that it only contains the libraries and services required by the application. Code minimization can be enforced at a much finer granularity by also minimizing the libraries so that they only contain the functionality needed by an application. For example, the generic C library, *libc*, is quite large but only a fraction of it is utilized by an application.

Our proposed design leverages multi-core architecture to achieve strong process isolation by executing each process on a separate set of dedicated processor cores. Each process executes on its customized OS image which is minimized to only contain the code (drivers, services etc.) required by the process. Each application provides a manifest file specifying its dependencies, that is, the list of libraries, services and drivers that this application needs to execute successfully. This manifest file serves as the starting point for dependency analysis that determines the external code required by a given application. For each application, we build a custom OS image consisting of a micro-kernel and the application along with its dependencies. This custom OS image will not have any extraneous service not required by the application.

This paper is organized as follows. In section 2, we present the high level design of the proposed architecture. In section 3, we present an analysis of *libc* library usage by popular applications. We then discuss the related work in section 4

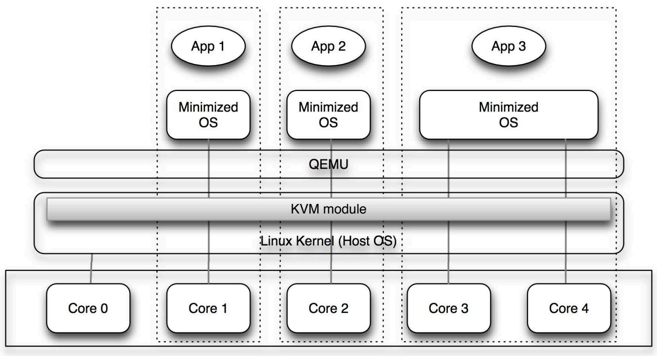

Figure 1: Overview of system architecture

and conclude in section 5.

2. SECURE ARCHITECTURE DESIGN

We propose a secure architecture design that leverages multicore architecture to achieve strong isolation guarantees between applications. Our basic architecture is inspired from the NoHype architecture [8] which uses multicore architecture to isolate virtual machines (VM) in a cloud setting. Our goal is to isolate not just guest VMs, but also the applications that run on it. We incorporate the basic NoHype architecture into our design and extend it according to our requirements. Figure 1 shows an overview of our system design.

In this architecture, each application is isolated by executing on its own dedicated VM that only executes this application and no other application. The isolation guarantees are further strengthened by enforcing isolation at the hardware layer. This is achieved by executing the application's corresponding VM on a dedicated set of processor cores that are not shared with other applications. At any given time, the system maintains a pool of unassigned processor cores. When a new application starts, it is assigned to a set of cores from this pool using the concept of *processor core affinity*. Memory isolation between VMs is enforced by capitalizing on the hardware paging mechanism in modern processors. Also, like NoHype, I/O devices are dedicated to each VM by virtualizing the I/O devices. Communications between applications executing on different cores is allowed via a restricted interface that is mediated according to certain predefined policies.

The guest OS that hosts an application is also minimized according to this application's requirement so as to reduce the software attack surface. That is, only the OS functionality required by an application is retained and parts of the OS that are not required by the application are removed. The OS functionality that is considered for elimination consists of unused kernel services, libraries, drivers and other utilities. This reduces the software attack surface that is available to an attacker to craft a potential exploit.

Our design assumes a large number of processor cores which is not unreasonable given the recent advances in multicore hardware. Commercial processors currently available in market already boast of more than 100 cores. The maximum number of cores required by a system is determined by the number of applications running at any given time and the number of cores required by each of them. For systems with limited numbers of cores, a variation of our proposed scheme can be used by which the applications are grouped according to a pre-defined criteria and the applications belonging to the same group are allowed to share processor cores. For example, applications written by the same developer can belong to the same group. This allows our scheme to function even with limited number of processor cores.

2.1 Communication

One important aspect of application isolation is to mediate the communication between two applications. Forbidding all communications between two applications is not always practical since they may need to communicate to request and provide services, exchange data etc. For example, in our design, services such as file server may run isolated on a VM and will need to communicate with other applications that require these services. Another example would be an email client that needs to invoke other applications based on email content, such as a PDF viewer or a web browser. While such interactions enhance functionality, they also represent an attack vector that can be exploited by malicious or infected applications. Thus, it is crucial that these interactions be monitored according to a set of policies.

The NoHype architecture has been developed for the cloud settings in which guest operating systems do not need to communicate with each other as they belong to different customers. This allows one to disengage the hypervisor from guest OS at runtime and eliminate the hypervisor attack surface. However, this is not true in our setting in which each guest OS is dedicated to a single application and applications typically need to communicate. To ensure proper isolation, the communication between the various applications needs to be mediated, by only allowing the authorized communications and preventing unauthorized interactions. Communication mediation can be enforced by the hypervisor by intercepting the communication and enforcing appropriate policies. This means that, unlike NoHype, the hypervisor cannot be disengaged from the guest OS because we need it to mediate the communication between applications.

We propose using RPC-like mechanism to facilitate communication between applications running on different guest VMs. The hypervisor can serve as a middleware for RPC calls and the policy enforcement point that allows or blocks communication. The hypervisor must provide a RPC interface that uniquely identifies the application and the core it is running on and is able to route the RPC calls correctly. As an example, a text editor application running on one core may need to read a file, but the file server is running on another core. The call to read the file gets converted into a RPC like call that is transmitted via the hypervisor to the core that is running file server and the response from the file server will follow the same route back. The hypervisor will also check whether the text editor is allowed to ask for file system services and whether it is authorized to read the requested file.

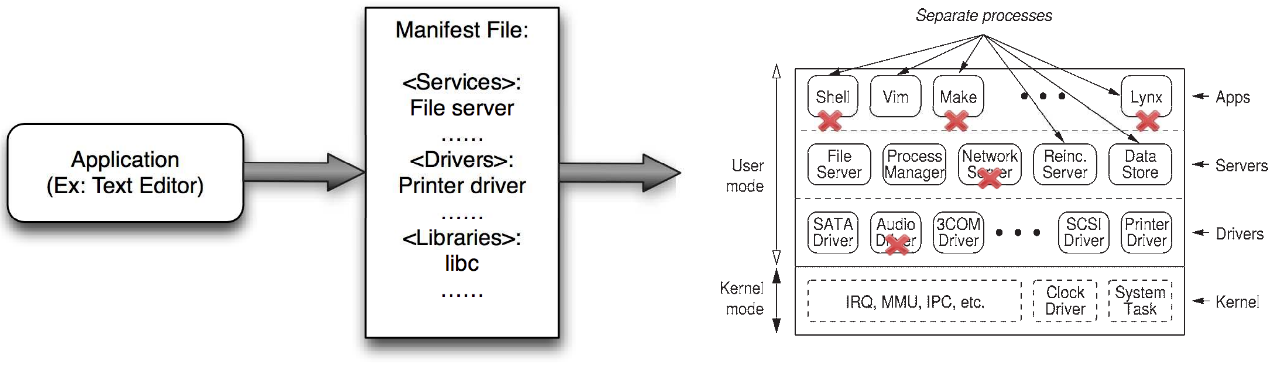

Figure 2: Example: Minimizing MINIX 3 Operating system for a text editor

2.2 Operating System Minimization

Traditional monolithic kernels violate the principle of least privilege by making several services and libraries available to an application even though the application does not require them. If the application is compromised, an attacker can misuse these system services for arbitrary code execution or privilege escalation attacks. To address this, the OS code must be minimized such that each guest OS only makes available the services required by the application executing on it.

To reduce the attack surface offered by the OS, one of the following approaches can be adopted. In the first approach, one can start with a monolithic kernel and remove the unwanted services, drivers and libraries from it. This would require removing a large chunk of code from the OS. The second approach involves starting from a microkernel and only adding the desired functionality that is required by an application. Instead of starting with a monolithic kernel and removing extra code, we adopted the microkernel approach which has only the basic functionality implemented in the kernel space.

We used MINIX 3, a microkernel based OS, for our prototype implementation. Figure 3 shows the design of MINIX 3 where various services (such as file server, network server) and drivers are implemented in the user space. Given an application's requirements, all the services and drivers not required by the application are removed from the OS and the required dependencies of the application are added. This can be viewed as creating a minimal standalone customized OS image for each application.

Figure 2 shows an example use case of minimizing the operating system according to application requirements. In this example, the application under consideration is a simple text editor that requires only few services provided by the MINIX 3 microkernel. For example, it does not need audio driver and network server and these should be removed from the OS to reduce attack surface. Also, MINIX 3 comes with some pre-installed applications such as Make, Lynx etc which should be removed if not required by the application (in this case, text editor).

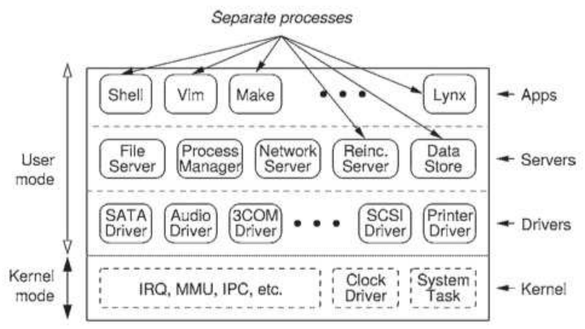

Figure 3: Architecture of MINIX 3 operating system (Photo credit - MINIX 3 document [4])

Dependency analysis

To generate a minimized custom OS image for an application, we need to obtain the list of drivers, services and libraries required by the application. These should be specified by the application developer in a manifest file that is included in the application package. Since these services may have their own dependencies, a transitive closure of all the dependencies required by the application must be constructed. Correctly specifying all the application dependencies, especially at kernel level, may require significant effort from an application developer. Various techniques can be used to simplify this task for the developer by automatically inferring some or all of these dependencies from either the source code or the executable file. For instance, by analyzing the source code, we can see the libraries included and the system calls made by the application and make some inference about system services used by an application. Process tracing is another approach that can be used to automatically infer application dependencies. This can be done by tracing the process execution and logging the system calls and API calls made by this application. In this approach, it is crucial to execute the application with all possible inputs to build a complete dependency profile for the application. Techniques like those used for software testing can be used to ensure that a complete coverage of inputs is achieved.

	LOC	Symbols
Total	276,970	2171
Min	131,653	360
Average	145,498	450
Std Dev	13,350	75

Figure 4: Statistics concerning lines of assembly code and symbols usage in *libc*

2.3 Library minimization

In addition to eliminating the libraries that are not needed by an application, a more fine grained minimization can be performed on the libraries themselves. In current systems, even if an application requires access to a single function in shared library, the entire library is mapped into the process memory image. This is a problem of over-provisioning of access. That is, the coarse granularity of shared library loading is inadequate for enforcing the *principle of least privilege* within the library. Consequently, an attacker exploiting an application vulnerability has more access to the library than is necessary. By minimizing the libraries so that only contain the required functionality, the principle of least privilege for code can be enforced more accurately.

To determine the library functionality that may be called by an application, we perform a control flow analysis to identify the reachable code in a library. This involves performing an any-path evaluation, beginning with the set of dynamically mapped symbols in the program's ELF symbol table. These can be retrieved using `objdump` utility. This any-path evaluation maintains a set of symbols to evaluate, where the set is initialized with the application's required symbols. Each symbol in this set is processed by traversing the corresponding binary instructions in a disassembled version of the shared library. Whenever any variation of `ret`, `jmp` or `call` instruction is encountered, the target symbol is added to the working set. This is executed until the least fixed point of symbol references is reached, indicating that all library instructions that could possibly be reached from the symbols in the application's symbol table have been traversed. The minimization technique only retains the code blocks of shared library corresponding to this set of symbols required by the application and removes rest of the code from the library.

3. *LIBC* USAGE ANALYSIS

To better understand typical *libc* usage in a variety of settings, we performed an empirical analysis of the usage of the *libc* library by 51 popular applications, which are listed in the Appendix. We selected applications from a broad range of categories (*e.g.,* network applications, language interpreters, virtual machine monitors, media applications). Our analysis is based solely on the binary executable, so we considered both open- and closed-source software. All software packages were downloaded for Ubuntu 10.04 ("Lucid Lynx") for version 2.6.32 of the Linux kernel.

To analyze an application's *libc* usage, we performed a control flow analysis as explained in section 2.3. In our static analysis, we discovered that there is a significant minimum threshold of *libc* code that must be retained for all applications. Specifically, all processes created from ELF executables jump to the `__libc_start_main` symbol, which indicates the start of the program. From this single symbol, every application reaches 360 of the 2171 function symbols in *libc*.

Figure 4 shows the total and minimum *libc* code base, as well as some basic statistics about the applications we surveyed. In Figures 5-6, the minimum values are labeled and marked with dotted line for reference. Figure 5 shows the number of *libc* symbols used by a subset of the applications we surveyed. For each application, the darker region on the left indicates the number of symbols explicitly identified by performing `objdump` on the executable; the lighter region on the right denotes the total number of symbols reached through our any-path analysis. Figure 6 shows the total number of *libc* lines of code that are required for each application. This figure shows that most applications only require about half the code that exists in *libc*. Furthermore, observe that most of this code is the minimum (*i.e.,* the code reachable from `__libc_start_main` entry point). From Figure 4, we observe that the average application requires only 90 additional symbols when compared with the minimum. That is, minimizing the *libc* code according to the application's needed functionality has a drastic effect on the attack surface. On average, the lines of code (*i.e.,* assembly instructions) and number of symbols used is significantly smaller than the full library size. In other words, mapping the full library gives the attacker access to a significant amount of unused code for attacking using attacks such as code-reuse attacks.

Finally, Figure 7 shows an interesting result concerning the frequency of non-minimal symbol usage among the 51 applications surveyed. That is, setting aside the 360 symbols derived from `__libc_start_main`, we calculated how many of the applications used each of the remaining symbols. For instance, there were 133 symbols that were used by only one application, while there was only one symbol used by 39 applications. Furthermore, 426 (72.2%) of the symbols in *libc* were used by 10 or fewer applications, while 522 (88.5%) of the symbols were used by 20 or fewer applications. In other words, there is significant variation in the non-minimal symbols that are used by applications. Consequently, the probability of any two applications using exactly the same set of symbols is very small; we observed only a single pair of applications (`gs` and OpenOffice) that used the exact same set of symbols, which happened to be the minimal set.

4. RELATED WORK

Our basic architecture is very similar to the NoHype [8] architecture as both leverage multicore to achieve isolation. However, our architecture has some important differences from NoHype as discussed below. First, the NoHype architecture has been designed specifically for cloud settings where the isolation is performed at VM level as opposed to application level as in our design. Second, NoHype uses the hypervisor only in the initial phase to allocate resources and in the end to terminate a VM. The hypervisor is disengaged from the VM when it starts the application execution. This is not the case in our design since in our design applications may need to communicate and constant involvement of hypervisor is required to mediate this communication. Also, unlike our approach, NoHype does not use code minimization for the guest OS running in the VM.

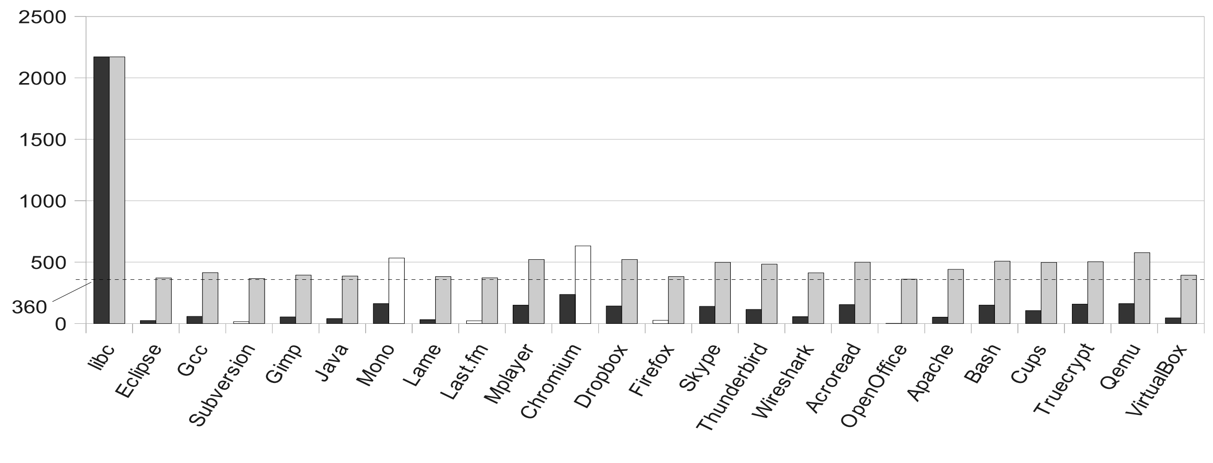

Figure 5: Number of *libc* symbols used by popular applications

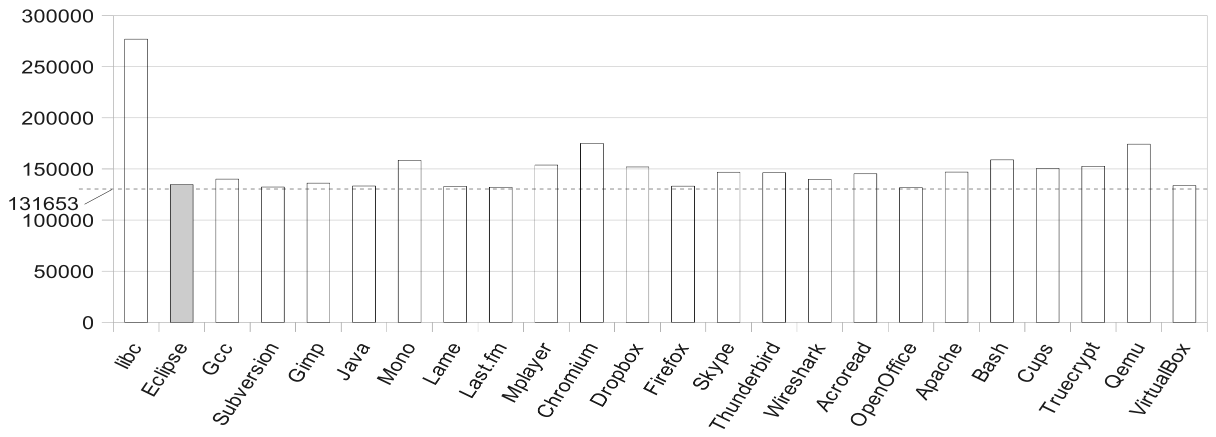

Figure 6: Number of lines of *libc* assembly code used by popular applications

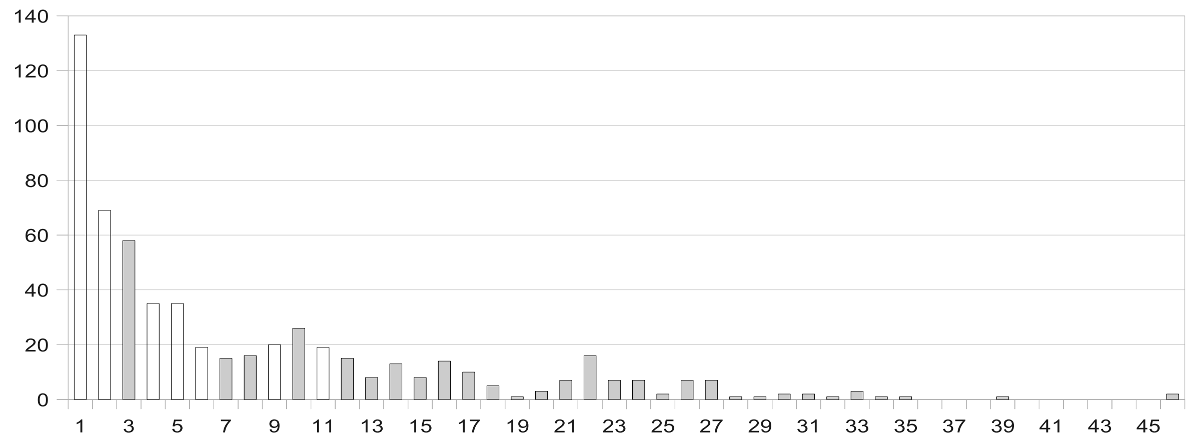

Figure 7: Histogram of non-minimum symbols used by popular applications

Poly^2 [1], a secure platform for network services, isolates network services on different systems and proposes minimization of operating system to remove unused services. However, this work is limited to network services and uses different machines for isolation. Our approach significantly extends Poly^2 by isolating all applications in addition to just network services. Also, in our architecture, the isolated applications still run on the same machine and isolation is enforced by eliminating sharing of the processor cores and isolating memory.

Codejail [10] is a framework to isolate untrusted libraries so as to prevent a bug in the library from compromising the main program. This is achieved by creating separate execution contexts, that is trusted and untrusted contexts for executing main program and untrusted library code respectively. All accesses to system resources from untrusted context are sandboxed. Codejail approach can be integrated into our proposed design to achieve a higher degree of isolation within an isolated VM.

Dong et al. [2] quantify security benefits and performance costs of performing privilege separation in web browser designs. In web browsers, various browser components can be grouped into isolated partitions that are assigned minimum privileges as needed for runtime operation. However, excessive isolation between browser components may increase communication between components and incur higher performance costs. [2] provides empirical data on this trade-off while choosing various partitioning strategies in web browsers.

Tartler et al. [9] reduce the size of trusted computing base by reducing the kernel code using compile time kernel configuration options. This reduces the attack surface available to the attacker. They showed a decrease of 10% in CVE vulnerabilities by reducing the kernel code using configuration option. This reduction is based on the anticipated workload and is limited to the KCONFIG features. On the other hand, our minimization is specific to the application that will execute on the guest OS.

5. CONCLUSION

In this work, we have proposed an initial secure architecture design that is based on the principles of application isolation and code minimization. The application isolation is enforced at both software and hardware layers to prevent a malicious application from compromising other parts of the system. The code minimization for the operating system code and the library code is performed to reduce the software attack surface that can potentially be exploited by an attacker. We have also performed analysis of *libc* library to show that only a fraction of the entire library is used by any given application. It is important to mention that to further securing applications from attacks such as ROP attacks, our architecture can be combined with the application randomization techniques by which the application code is randomized at each execution [3]. As result, security will be achieved by the combination of four key approaches: isolation, mediated communication, minimization, and randomization.

6. REFERENCES

[1] E. Bryant, J. Early, R. Gopalakrishna, G. Roth, E. Spafford, K. Watson, P. William, and S. Yost. Poly2 paradigm: a secure network service architecture. In *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*, pages 342–351, 2003.

[2] X. Dong, H. Hu, P. Saxena, and Z. Liang. A quantitative evaluation of privilege separation in web browser designs. In *ESORICS*, 2013.

[3] A. Gupta, S. Kerr, M. Kirkpatrick, and E. Bertino. Marlin: A fine grained randomization approach to defend against rop attacks. In J. Lopez, X. Huang, and R. Sandhu, editors, *Network and System Security*, volume 7873 of *Lecture Notes in Computer Science*, pages 293–306. Springer Berlin Heidelberg, 2013.

[4] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Minix 3: a highly reliable, self-repairing operating system. *SIGOPS Oper. Syst. Rev.*, 40(3):80–89, July 2006.

[5] S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 143–157, 2012.

[6] Kostya Kortchinsky. CLOUDBURST: A VMware Guest to Host Escape Story. `http://www.blackhat.com/presentations/bh-usa-09/KORTCHINSKY/BHUSA09-Kortchinsky-Cloudburst-SLIDES.pdf`.

[7] National Vulnerability Database (NVD). Vulnerability Summary for CVE-2009-1244. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-1244`.

[8] J. Szefer, E. Keller, R. B. Lee, and J. Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 401–412, New York, NY, USA, 2011. ACM.

[9] R. Tartler, A. Kurmus, B. Heinloth, V. Rothberg, A. Ruprecht, D. Dorneanu, R. Kapitza, W. Schröder-Preikschat, and D. Lohmann. Automatic os kernel tcb reduction by leveraging compile-time configurability. In *Proceedings of the Eighth USENIX conference on Hot Topics in System Dependability*, HotDep'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.

[10] Y. Wu, S. Sathyanarayan, R. H. C. Yap, and Z. Liang. Codejail: Application-transparent isolation of libraries with tight program interactions. In *ESORICS*, pages 859–876, 2012.

[11] Xen Security Advisory CVE-2012-0217. 64-bit PV guest privilege escalation vulnerability. `http://lists.xen.org/archives/html/xen-announce/2012-06/msg00001.html`.

APPENDIX

A. APPLICATIONS SURVEYED

Application	Version
Acroread	9.3.2
Amazon MP3 Downloader	1.0.9
Apache	2.2.14
Bash	4.1.5
Bluetooth	4.60
Brasero	2.30.2
Chromium	10.0.648.205
Cups	1.4.3
Dhclient	3.1.3

Dropbox	0.6.7
Eclipse	3.5.2
Empathy	2.30.3
Evolution	2.28.3
Firefox	3.6.16
Gcc	4.4.3
Gimp	2.6.8
Git	1.7.0.4
Gnome-terminal	2.30.2
Grip	3.3.1
Gs	8.71
Gtkpod	0.99.14
Gzip	1.3.12
Java	1.6.0_20
Lame	3.98.2
Last.fm	1.5.4
libc	2.11.01
Make	3.81
Mencoder	4.4.3
Mono	2.4.4
Mplayer	1.0
OpenOffice	3.2.0
OpenSSH	5.3p1
PdfTeX	1.40.10-2.2
Perl	5.10.1
Python	2.6.5
Qemu	0.12.3
Sha1sum	7.4
Skype	2.1.0.81
Smbclient	3.4.7
Subversion	1.6.6
Sudo	1.7.2p1
Tar	1.22
Thunderbird	3.1.8
Totem	2.30.2
Truecrypt	6.3a
Vim	7.2
VirtualBox	4.0.0
Vlc	1.0.6
Transmission	1.93
Wine	1.2.2
Wireshark	1.2.7
Xine	0.99.6