Attack Surface Reduction For Commodity OS Kernels

Trimmed garden plants may attract less bugs

Anil Kurmus
kur@zurich.ibm.com
IBM Research - Zurich

Alessandro Sorniotti
aso@zurich.ibm.com
IBM Research - Zurich

Rüdiger Kapitza
rrkapitz@cs.fau.de
Friedrich-Alexander University
Erlangen-Nuremberg

ABSTRACT

Kernel vulnerabilities are a major current practical security problem, as attested by the weaknesses and flaws found in many commodity operating system kernels in recent years. Ever-growing code size in those projects, due to the addition of new features and the reluctance to remove legacy support, indicate that this problem will remain a severe system security threat in the foreseeable future. Reactive measures such as bug fixes via code reviews and testing, while effective, can only alleviate the issue. Furthermore, common practices in system hardening often focus on complex and sometimes hard to achieve goals that require extensive manual intervention such as security policies for sandboxing.

In this paper, we explore an alternative, automated and effective way of reducing the attack surface in commodity operating system kernels, which we call *trimming*. Trimming is a two-fold process: an initial analysis of a given system for unused kernel code sections is followed by an enforcement phase, in which the unused sections are removed or prevented from being executed. We discuss the requirements that should be reflected in the design of a trimming infrastructure, and present a lightweight and flexible implementation example for the Linux kernel by using dynamic binary instrumentation as provided by *kprobes*. Our evaluations show we can, in the case of a web server, reduce the attack surface of the kernel (in terms of the number of kernel functions accessible from unprivileged users) by about 88%.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Kernel security; K.6.5 [**Security and Protection**]: Unauthorized access

General Terms

Reliability, Design

Keywords

Attack Surface Reduction, Kernel Hardening

EUROSEC '11 April 10, Salzburg, Austria

1. INTRODUCTION

Large software development projects, such as the Linux kernel, suffer from feature explosion [1], which leads to an ever-growing code size. This problem is also aggravated by backwards-compatibility requirements. While good software projects are structured in such a way that the feature-explosion problem does not impact performance, the presence of additional privileged code, which is always the case for additional code in monolithic kernels, will increase the attack surface of a system and reduce the overall security of the software. This is well illustrated by recent local privilege escalation exploits for the Linux kernel: e.g., in the `vmsplice` system call (CVE-2008-0009, CVE-2008-0010, CVE-2008-0600), the RDS sockets (CVE-2010-3904), `v4l1_compat` ioctl (CVE-2010-2963), or `NULL` pointer dereferences in various sockets (CVE-2009-2692, CVE-2009-2698).[1]

The increasing attention given to kernel security is also explained by the improvements in the mitigation of user-space vulnerabilities: because of methods such as address space layout randomisation, non-executable data pages, stack smashing protections, and, recently, gadget-free binaries [2], the kernel now compares to privileged user-space applications as an easier target for attackers. While reactive measures, such as code reviews, and the use of source code scanners and fuzzers can improve code quality and reduce the number of weaknesses in a software, some will still remain, and proactive measures are necessary.

Most of the aforementioned vulnerabilities could be prevented from being exploited by using known approaches (e.g., the `mmap_min_addr` check, grsecurity PaX randkstack, kernexec, uderef patches). However, we observe that all reside in parts of the Linux kernel that are not commonly used in production systems, and could be disabled through automatical removal of unnecessary code paths from the kernel.

Accordingly, we present *trimming*, an approach for reducing the attack surface in commodity operating system kernels for a given set of applications and services. Trimming is performed in two consecutive phases: an initial analysis for unused kernel code sections that is followed by an enforcement phase, in which the unused sections are removed or prevented from being executed. The target of such a mechanism is a system with well-defined use cases, typically a server performing a specific task (e.g., web server, database, network-attached storage, or soft-router). This allows us to make the assumption that the kernel features used by such a system do not change significantly over time. We believe such an assumption to be realistic in the case of produc-

[1] `http://cve.mitre.org/`

tion servers, or virtual appliances. Our threat model is a local unprivileged attacker, who might have obtained his or her access through a remote vulnerability in a service, and attempts to exploit a kernel vulnerability, for example for privilege escalation, or a kernel crash. We implemented a lightweight trimming infrastructure for the Linux kernel provided as loadable module that uses *kprobes* [3]. The latter enables us to dynamically instrument function calls inside the kernel and can be used for the analysis phase as well as for the enforcement phase. Our initial evaluations are promising, we can reduce the reachable kernel functions for a web server by approximately 88%. While the overhead due to dynamic instrumentation is small for our macro-benchmarks — it varies between 0-15% for instrumenting 20% of all non-static kernel functions — we discuss future steps to reduce the runtime instrumentation overhead.

The remainder of the paper is structured as follows: Section 2 details requirements and the overall approach. Section 3 outlines our initial prototype and discusses first evaluation results. In Section 4, we briefly compare related approaches and finally conclude in Section 5.

2. GENERAL APPROACH

Before we detail the two-phased process of trimming a kernel we outline the general goals that should be met by a concrete implementation of the process.

2.1 Goals

We identify the following requirements for an ideal *trimmed* kernel:

1. *effectiveness*: in order to reduce the kernel attack surface, trimming should disable as much unused code as possible.
2. *stability*: the trimmed kernel must remain stable under common-case usage. More precisely, any non-malicious operation crashing the trimmed kernel should also crash the original kernel; trimming should not break any user-space application that was expected to be used with the trimmed kernel.
3. *performance*: low or no common-case performance overhead. In some cases, performance improvements can even be expected (due to reduced memory footprint of the kernel).
4. *security*: trimming should not introduce new vulnerabilities in the kernel or user-land. Note that, some malicious operations (i.e., exploits) may have their impact (e.g., privilege escalation) mitigated by degrading them into denial of service attacks for the application in the case of a trimmed kernel: we do not consider this as the introduction of a new vulnerability. Additionally, it should not be possible to circumvent trimming: if an access control mechanism is used in order to prevent unused code from being executed, it should not be possible to bypass it (in particular, it should not be vulnerable to time-of-check-to-time-of-use races).
5. *usability*: trimming should only require minimal manual setup (e.g., no policies to write), and have the possibility for privileged users to create, override or adapt trimming, preferably without rebooting. Ideally, loadable kernel modules should also be trimmed.

2.2 Generic system design

In this section, we propose a generic system design for kernel trimming. We distinguish two phases: an initial *analysis* phase, followed by an *enforcement* phase.

2.2.1 Analysis

In the analysis phase, the kernel, and the applications it should run, are automatically analysed in order to distinguish required code sections from unused ones. This requires that the analysed system is not compromised. This step can be performed either by static or dynamic analysis. The granularity considered for code sections while performing analysis can vary: instruction granularity, basic-code-block granularity, function granularity, or any given subset of kernel functions (e.g., kernel functions exported for use by loadable kernel modules, system calls).

A static source code analyser can, beside other things, identify dead-code or unreachable code [4]. However, it can be argued that many applications include a large amount of functionality that is disabled at runtime (e.g., by reading a configuration file), and the effectiveness of such a static approach would be lower.

A dynamic analysis phase is essentially a learning phase: the applications are subjected to non-malicious operations, simulating their expected use of the kernel. By dynamically tracking which code paths are used by the application, we can generate a white-list of allowed kernel code.

2.2.2 Enforcement

The enforcement phase aims to prevent execution of kernel code sections that have not been white-listed in the analysis phase. This can be achieved by *removing* or *disabling* the relevant code blocks. Removing code (e.g., by recompiling the kernel) has the advantage of creating a smaller kernel, which can improve performance. Alternatively, disabling code, typically by adding an access control check at the beginning of the code section, has the advantage of providing more flexibility at runtime: various applications can access different parts of the kernel.

In both cases, modifications to the kernel are required, whether they affect the kernel source code, kernel disk image, or in-memory kernel image. In all the latter cases, we remark that static or dynamic binary instrumentation tools are well suited for adding access control checks.

3. AN IMPLEMENTATION FOR LINUX KERNELS: KTRIM

3.1 System description

In this section, we describe the design of a proof-of-concept kernel trimming infrastructure for Linux implemented as a loadable kernel module. The module performs the analysis and enforcement phases dynamically by using *kprobes* [3] for dynamic binary instrumentation. Kprobes exports an interface for kernel modules to register a *probe* for a given kernel code address and a handler function that is called when the code at the given address is executed.

3.1.1 Setup phase

In our implementation, we add a setup phase to the generic system design, where we select a set of kernel functions that should be probed, i.e. intercepted, in the analy-

sis phase. For our tests, we have tested two lists: a small set of all system calls (around 350 functions), and a larger set of around 2k functions selected out of a total of around 10k non-static kernel functions. In our proof-of-concept, we refrained from using more functions (including static ones) due to performance and stability considerations. A too finely granular approach would indeed be more likely to have false-positives in the enforcement phase. Also, as a technical note, registering a probe in our handler or any function called by our handler would result in unbounded recursion and kernel crash, therefore those functions should not be probed. The large set was built by taking functions which were exported for use by non-GPL modules, as these symbols usually correspond to distinct functionality provided by the kernel. Although our tests show that this initial 2k-set provides satisfactory code coverage effectiveness, stability and performance, we acknowledge that the set of functions would certainly benefit from being chosen, in future work, by a more systematic approach (e.g., by using call graphs derived from the kernel source code or binary).

We also select a *policy scope*, which is essentially the set of applications that should be analysed and for which enforcement should take place. In other words, these are the applications that are likely to be under the control of an attacker willing to take advantage of a kernel vulnerability. Presumably, in the absence of additional hardening (i.e., MAC type enforcement), there is no valid security reason for including applications that will run with system privileges in the policy scope, because an attacker compromising such an application will most likely not need to exploit a kernel vulnerability to achieve his goals. In our implementation, the policy scope is restricted to tasks without the `CAP_SYS_ADMIN` POSIX capability (which is equivalent to root). However, a policy scope can also be defined by processes executed on behalf of a certain user, or a security context (for systems with MAC support), or a *cgroup*[2].

3.1.2 Analysis phase

In the analysis phase, we essentially run applications in the *policy scope* under non-malicious user inputs in order to learn which kernel functions will be used. This is done by inserting analysis probes on kernel functions selected in the setup phase which are associated with the handler routine in Algorithm 1. Whenever a probe is reached in the context of an application in the policy scope, the associated handler code is run: in this case, the probe is removed from the list of active probes. At the end of the analysis phase, all remaining active probes correspond to kernel functions that are not used by applications in the policy scope. This constitutes the system's *trimming policy*.

Algorithm 1 Analysis-probe handler algorithm

ANALYSIS-PROBE-HANDLER(*current-task*)
  if *current-task* ∈ *policy-scope*
    $probes = probes \setminus \{current\text{-}probe\}$

[2]See Linux kernel documentation: `Documentation/cgroups/cgroups.txt`.

3.1.3 Enforcement phase

The enforcement phase starts by replacing the handler of all probes in the trimming policy from the analysis phase, with the enforcement-probe handler (see Algorithm 2). When this handler is triggered by an application in the policy scope, kernel code that was not used in the analysis phase is being run, therefore the KTRIM-ENFORCE-DENIAL function is called to act on this event. For example, in order to prevent a potential attack, the infringing application is killed by this function, and the attempt logged. Note that this means both analysis and enforcement only occurs on code running in process context, as we do not have any user to blame for code running in interrupt context.

Algorithm 2 Enforcement-probe handler algorithm

ENFORCEMENT-PROBE-HANDLER(*current-task*)
  if *current-task* ∈ *policy-scope*
    KTRIM-ENFORCE-DENIAL(*current-task*)

3.2 Evaluation and discussion

In this section, we evaluate ktrim against the requirements identified in Section 2.1. In case of shortcomings, we discuss possible improvements.

Our evaluation setup and workload consists of benchmark tests using the Phoronix test suite[3]; the test suite is run on a VirtualBox VM with 500 MB of RAM running Fedora 14. Among the various tests included in the suite, we select six that specifically test usual operations carried out within the chosen use-cases for ktrim i.e. web server, databases and so forth. For each selected test, we perform three separated runs: one without the ktrim module (Reference); one with the ktrim module loaded and in the analysis phase (Analysis); a last one with the module loaded and in the enforcement phase (Enforcement). For each test, we consider two different function sets (the 2k-exported-symbol-set and the system-call set), as explained in Section 3.1.1.

3.2.1 Effectiveness

We measure the effectiveness of this implementation by estimating the amount of (process context) kernel code that is prevented from being executed for applications in the policy scope. In the case where we use the large set of exported symbols, since this set is a representative sample of all non-static kernel functions (about 20%), we can estimate the attack surface reduction by using the ratio of the number of probes remaining in the enforcement phase to the total number of probes. In the case of the apache benchmark, 227 symbols are kept out of a total of 1922. This represents an attack surface reduction of around 88%, which shows this approach is very effective.

Effectiveness can be further improved by providing support for multiple policy scopes. By analysing various applications separately, we can enforce accesses to different code sections in the kernel, and thus reduce the attack surface for an attacker in a given policy scope. However, this can increase the overhead of trimming.

[3]`http://www.phoronix-test-suite.com/`

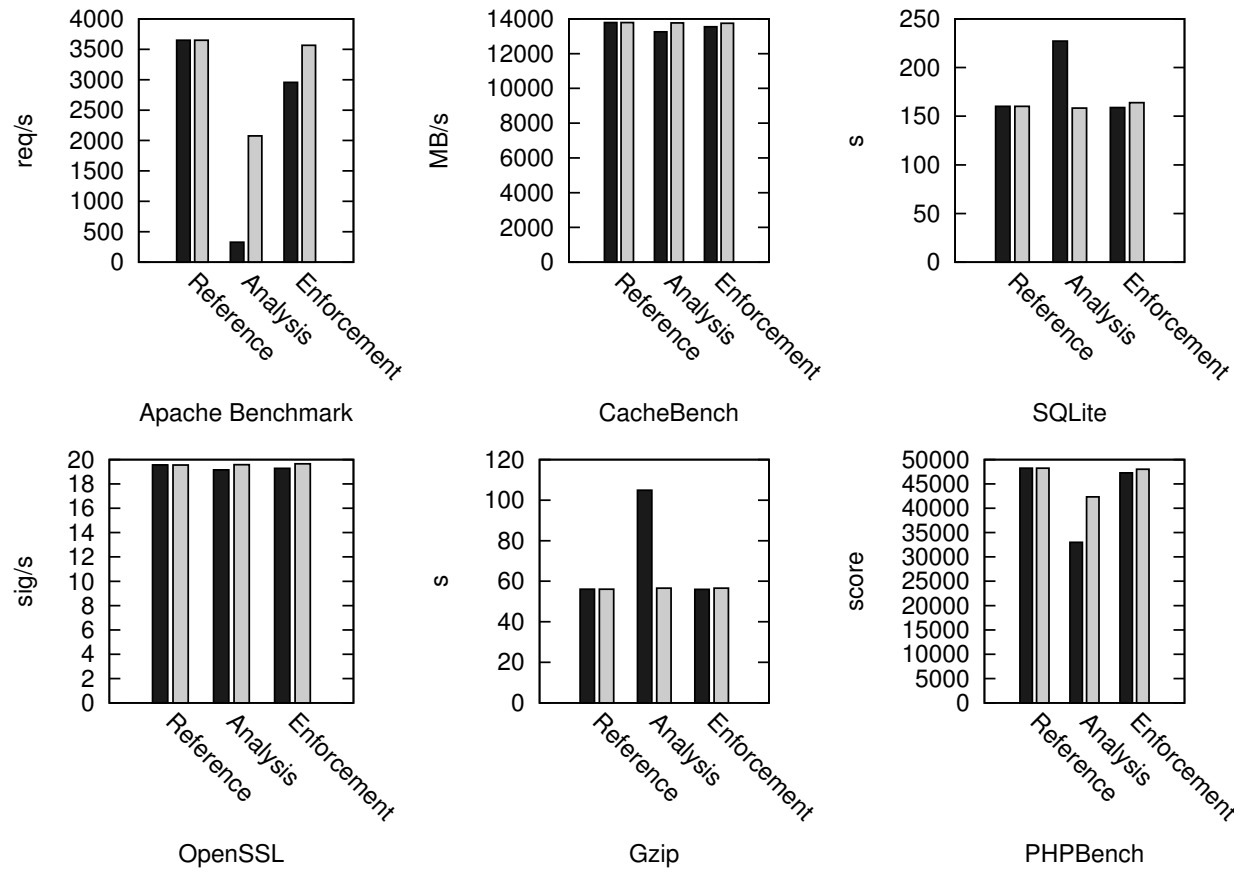

Figure 1: Performance results. Dark bars represent the 2k-exported-symbol set. Light bars represent the system-call set.

	system-call set (342 probes)	2k-symbol set (1922 probes)
Apache	37 (89%)	227 (88%)
CacheBench	18 (95%)	136 (93%)
Gzip	35 (90%)	131 (93%)
OpenSSL	18 (95%)	72 (96%)
PHPBench	18 (95%)	72 (96%)
SQLite	18 (95%)	108 (94%)

Table 1: Number of probes removed in the analysis phase for each benchmark (in parenthesis, attack surface reduction)

3.2.2 Stability

As expected, under our test workload, the kernel with the ktrim module did not crash. Concerning the behaviour of user-space applications, we note that it is important that the analysis phase is run over a long enough time to capture all possible sporadic or periodic events (e.g., cron jobs), in order to maintain their expected functionality. Alternatively, the use of static analysis (in the kernel and the applications) in the analysis phase can help to deduce all code sections required by the applications. However, it can be argued that many applications include a large amount of functionality that is disabled at runtime (e.g., by reading a configuration file), and the effectiveness of such a static approach would be lower.

We also remark that for some applications, killing the process when enforcing might break their functionality. For example, in the case of a multi-user application such as a web server which is threaded with user-level threads (or uses a single thread with non-blocking I/O), killing the process will necessarily result in the loss of service of all users, including non-malicious ones, which is undesirable. More fine-grained possibilities for the KTRIM-ENFORCE-DENIAL function include interactions with the user-space to enforce the policy infringement. Namely, the task can be put into uninterruptible sleep while signalling the breach to a helper application that will terminate only the session of the infringing user before restoring the state of the application to resume on another user's session.

3.2.3 Performance

Figure 1 shows performance results of our tests: each histogram in the array represents the result of a test; the lighter bars represent the smaller set of all system calls, whereas the darker bars represent the wider set of 2k functions. From the figure we notice that the results of the tests without the module and with the module in enforcing mode are practically identical: this is a very desirable result, as in common-case usage a hardened machine would run the ktrim module in enforcement mode with very limited additional cost (the highest penalty paid is around 15% overhead on tests with

a high density of operations in kernel mode, the smallest is less than 1%). We also notice that the analysis phase is the one with the highest overhead: however i) this phase is supposed to be limited in time and to be run only once and ii) it is very expensive only on tests that require a lot of operations from the kernel: for example, the apache benchmark — which is I/O and kernel-heavy — is almost 10 times as slow.

It is clear that performance could be greatly improved with the use of static instrumentation for the enforcement phase. However, we also believe that we could use a faster dynamic instrumentation technique than kprobes by tailoring it to our use case (while kprobes for example uses jumps instead of breakpoints, it performs some unnecessary operations in our case, such as some register saves).

3.2.4 Security

We note that the enforcement phase, which is security critical, depends solely on the policy scope and access-control relevant properties of the current task (e.g., user id, capabilities, security context, cgroup). None of these variables are meant to be under the control of an unprivileged user, hence enforcement is only possible to bypass if ktrim and/or kprobes contain implementation vulnerabilities. We also point out that our implementation is very simple, around 300 lines of kernel module code, which makes it easy to audit and unlikely to contain any vulnerabilities.

3.2.5 Usability

Ktrim is designed as a kernel module that can be controlled through the `/proc` pseudo-filesystem. A system administrator can change phases (enforcement, analysis) or disable ktrim at anytime. This can be useful for an administrator willing to quickly update the trimming policy on the system to adapt to new changes, typically in a testing phase before the application is rolled out in a service. However, because the analysis phase requires a system that was not compromised, these modifications should be performed carefully. Additional usability improvements could be made by allowing to add or remove new probes or change the policy scope dynamically (i.e., a dynamic setup phase). However we believe this can negatively impact security and stability.

4. RELATED WORK

We identified three areas of related research that will be discussed in the following. We explicitly excluded architectural solutions such as microkernels because trimming targets shortcomings of current commodity operating systems.

4.1 Loadable kernel modules

Loadable kernel modules (LKMs) allow monolithic kernels to dynamically alter kernel functionality. This is used for example in Linux distributions to only load required drivers, filesystems, and networking services and protocols. In a sense, LKMs already achieve many of the trimming goals we list in Section 2.1. However, we achieve better granularity when discerning among code sections (i.e., better efficiency): as an example an application might only need a particular subset of filesystem operations, which LKMs cannot discern, but trimming does. Clearly, our implementation of trimming also provides the ability to disable an unnecessary module entirely for applications within policy scope, as long as the relevant symbols of the module were included in the setup phase. More importantly, LKMs are not used with the goal of reducing the attack surface of attackers, but mostly to reduce the memory footprint of a running kernel. This is well illustrated in Linux by the fact that most kernel modules will be automatically loaded when any (unprivileged) user-land application requires them, which means that the attack surface of a local attacker on the kernel includes all the automatically loadable kernel modules. Similarly, in order to reduce the footprint of the kernel without focusing on security, *specialisation* [5] is an approach used on embedded systems to fit the kernel to the applications it is running.

4.2 Sandboxes

Sandboxes based on system call interposition [6–10] and mandatory access control (MAC) mechanisms [11–13] provide the possibility to whitelist permissible operations for selected applications by creating a security policy. Sandboxes also reduce the attack surface of an attacker targeting the kernel, as the policy will restrict the access to some kernel code. However, this effect is secondary, as sandboxes aim primarily to prevent applications from performing security sensitive operations (e.g., `open(/etc/shadow,O_RDONLY)`) and not operations that can only result in privilege elevation when their implementation is vulnerable (e.g., the `vmsplice` system call). On the contrary, trimming aims to prevent access to unnecessary code sections in the kernel regardless of the fact that they semantically correspond to additional privileges for the application; it is especially powerful to prevent the exploitation of kernel code, but much less against the access of resources that become sensitive because of their relevance to the user-land (e.g., trimming cannot distinguish between `open(/etc/shadow,O_RDONLY)` and `open(/etc/passwd,O_RDONLY)`). Therefore, we see sandboxing and trimming as two complementary approaches that can be jointly used to achieve better system hardening.

We also note that the seccomp sandbox[4] in *mode 2* sandboxes processes by preventing a subset of system calls from being executed, achieving a similar functionality as our setup based on the small set of kernel symbols. In order to achieve better granularity, seccomp also uses *filters* on the system call arguments, in order to further restrict system calls depending on their arguments. Filters can be manually specified in the policy, together with the bitmap of allowed system calls. We believe however that this is a disadvantage: filtering system call arguments prevents simple automation of the policy creation process (analysis phase), although this is possible to achieve probabilistically, as shown in [14]. Additionally, the manipulation of the system call arguments introduces possible races and other vulnerabilities [15]. In other words, we believe that this conflates the role of the sandbox, thereby achieving a middle ground between the two opposite goals of user-land permission containment and attack surface reduction, as explained in the previous paragraph.

4.3 Hypervisor-based reference monitors

Virtual machine introspection, as introduced in [16], and virtualisation in general is an attractive technology for providing security guarantees on a guest kernel, as it allows external monitoring of a guest for events such as memory access. This makes it possible for example to perform rootkit

[4] `http://code.google.com/p/seccompsandbox/wiki/overview`

detection [17] with kernel memory integrity and control-flow integrity checks, even when a guest is entirely compromised [18], which is clearly not possible for a self-monitoring kernel. As a particularly original example, Chen, et al., [19] propose an approach to get around the weaknesses of commodity operating system kernels by protecting selected applications within the guest from an untrusted guest kernel, with the use of encrypted and authenticated memory pages.

We acknowledge that trimming could also be performed through hypervisor-based monitors, by using active hooks such as those provided by the Lares framework [20], or, for less overhead with in-VM monitoring [21]. However, we currently see no necessity for a hypervisor-based approach and in cases where resources are scarce, like soft-routers, this adds too much overhead, while in already virtualised environments, like infrastructure clouds integration might be difficult without vendor support.

5. CONCLUSION

There is a recent trend, partly driven by the ongoing improvements in the proactive mitigation of user-space vulnerabilities, to focus on attacking the kernel. One important source of flaws and weaknesses in the kernel is legacy code that in many cases is not actively accessed in normal operation mode but only in the context of an exploit. Taking these facts into account, we propose a kernel-trimming process that is composed of an initial analysis of a given system for unused kernel code sections and an enforcement phase in which unexpected control flows are intercepted. Our initial evaluation results based on an early prototype that performs dynamic instrumentation of kernel functions are promising. We were able to reduce the kernel attack surface for a web server considerably with only small performance penalties. Based on these results we plan to widen our empirical studies by evaluating further network-based services, designing a more precise attack surface metric, and improving the instrumentation framework, e.g., by evaluating the use of static analysis techniques.

6. REFERENCES

[1] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat, "Feature Consistency in Compile-Time Configurable System Software," in *Proceedings of the EuroSys 2011 Conference (EuroSys '11)*, 2011.

[2] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries," in *Proceedings of the 2010 Annual Computer Security Applications Conference*, 2010.

[3] A. Mavinakayanahalli, P. Panchamukhi, J. Keniston, A. Keshavamurthy, and M. Hiramatsu, "Probing the guts of kprobes," in *Proceedings of the 2006 Linux Symposium*, 2006.

[4] P. J. Guo and D. Engler, "Linux kernel developer responses to static analysis bug reports," in *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, 2009.

[5] D. Chanet, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere, "System-wide compaction and specialization of the linux kernel," *ACM SIGPLAN Notices*, 2005.

[6] N. Provos, "Improving host security with system call policies," in *Proceedings of the 12th conference on USENIX Security Symposium*, 2003.

[7] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A secure environment for untrusted helper applications confining the wily hacker," in *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography*, 1996.

[8] A. Dan, A. Mohindra, R. Ramaswami, and D. Sitaram, "Chakravyuha: A sandbox operating system for the controlled execution of alien code," tech. rep., IBM TJ Watson research center, 1997.

[9] A. Acharya and M. Raje, "MAPbox: using parameterized behavior classes to confine untrusted applications," in *Proceedings of the 9th conference on USENIX Security Symposium*, 2000.

[10] C. Willems, T. Holz, and F. Freiling, "Toward Automated Dynamic Malware Analysis Using CWSandbox," *IEEE Security and Privacy*, 2007.

[11] SELinux. `http://selinuxproject.org/page/`.

[12] TOMOYO. `http://tomoyo.sourceforge.jp/`.

[13] grsecurity. `http://grsecurity.net/`.

[14] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel, "Anomalous system call detection," *ACM Transactions on Information and System Security*, pp. 61–93, February 2006.

[15] T. Garfinkel, "Traps and pitfalls: Practical problems in system call interposition based security tools," in *Proceedings of the Network and Distributed Systems Security Symposium*, 2003.

[16] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proceedings of the Network and Distributed Systems Security Symposium*, 2003.

[17] N. Petroni Jr, T. Fraser, J. Molina, and W. Arbaugh, "Copilot-a coprocessor-based kernel runtime integrity monitor," in *Proceedings of the 13th conference on USENIX Security Symposium*, 2004.

[18] M. Christodorescu, R. Sailer, D. L. Schales, D. Sgandurra, and D. Zamboni, "Cloud security is not (just) virtualization security: a short paper," in *Proceedings of the 2009 ACM workshop on Cloud Computing Security*, CCSW '09, 2009.

[19] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems," in *Proceedings of the 13th international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, 2008.

[20] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.

[21] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-vm monitoring using hardware virtualization," in *Proceedings of the 16th ACM conference on Computer and Communications Security*, CCS '09, 2009.