Dynamic Integrity Measurement and Attestation: Towards Defense Against Return-Oriented Programming Attacks

Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy
Horst Görtz Institute for IT Security
Ruhr-University Bochum, Germany
lucas.davi@rub.de, ahmad.sadeghi@trust.rub.de, marcel.winandy@trust.rub.de

ABSTRACT

Despite the many efforts made in recent years to mitigate runtime attacks such as stack and heap based buffer overflows, these attacks are still a common security concern in today's computing platforms. Attackers have even found new ways to enforce runtime attacks including use of a technique called return-oriented programming. Trusted Computing provides mechanisms to verify the integrity of all executable content in an operating system. But they only provide integrity at load-time and are not able to prevent or detect runtime attacks. To mitigate return-oriented programming attacks, we propose new runtime integrity monitoring techniques that use tracking instrumentation of program binaries based on taint analysis and dynamic tracing. We also describe how these techniques can be employed in a dynamic integrity measurement architecture (DynIMA). In this way we fill the gap between static load-time and dynamic runtime attestation and, in particular, extend trusted computing techniques to effectively defend against return-oriented programming attacks.

General Terms

Measurement, Security

Keywords

return-oriented programming, integrity monitoring, attestation systems

1. INTRODUCTION

Distributed computing and worldwide business transactions over open networks, such as the Internet, increasingly demand for secure communication and secure operation due to rising online fraud [13] and software attacks [30]. While cryptographic techniques protect data communication satisfactorily in a pragmatic view, the security of the endpoints and their underlying software components suffer from exploitation of different vulnerabilities. Some of these vulnerabilities are due to the complexity and architectural constraints of the underlying execution environment (CPU hardware and commodity operating systems), some are due to poor software development practices and lack of software security in applications. In this context, the integrity of system software and applications is a fundamental requirement and necessary consequence in order to ensure trust in the computing infrastructure.

Trusted Computing as proposed by the Trusted Computing Group (TCG) offers a technology that is able to verify the integrity of executable content through remote attestation. This procedure allows a verifier to check the integrity of a remote system by verifying integrity measurement values digitally signed by a trusted hardware component called Trusted Platform Module (TPM) [35]. Several operating system extensions [22, 29] already support the TPM as underlying security module. However, such attestation mechanisms provide only integrity verification at load-time but not at run-time: An attacker can change the flow of execution of a program, e.g., via buffer overflow attacks [2] that are despite numerous countermeasures still a great security concern in software systems today.

In particular, new runtime attacks were found based on so called *return-oriented programming* [5, 31]. These attacks do not need to inject new code, but instead use code that already exists in the process's memory. Existing protection mechanisms such as marking the stack as non-executable [24] cannot detect this class of attacks because only instructions are executed that reside in valid code pages. Moreover, the new attacks generalize the original return-into-libc attack [32] by allowing the attacker arbitrary computation without calling any functions. In a traditional return-into-libc attack an attacker could execute only straight-line code without using branching, and could only invoke functions that reside in libc. In the new attacks, an attacker overwrites the stack with return addresses that point to existing code fragments in the program or system libraries.

The existing TCG attestation method cannot reflect such malicious changes during runtime since it takes static integrity measurements at load-time. Recent works try to fill the gap between load-time and runtime attestation by monitoring (dynamic) properties of programs [16, 12] or by enforcing information flow policies that allow low integrity data to flow only to high integrity processes [14]. However, these approaches require either the source code of applications to be monitored or are restricted to special runtime environments. To the best of our knowledge, none of these approaches can detect or prevent attacks that are based on return-oriented programming, which we consider a major threat in practical computing environments.

We propose a dynamic integrity measurement technique that is able to detect this new class of attacks. Our solution extends existing load-time measurement with a novel dynamic integrity monitoring using code rewriting techniques. In this paper, we present the current work-in-progress of our dynamic integrity measurement architecture (DynIMA). In particular, our contributions are the following:

- We propose a general design for a dynamic integrity measurement architecture (Section 3). The main idea is to instrument the code of programs before loading them to include runtime checks. These checks monitor changes during runtime in the data segment, in particular on the stack in order to detect attacks such as return-oriented programming.

- We explore two promising rewriting techniques to integrate in our integrity measurement architecture as tracking instrumentation (Section 4): taint tracking and dynamic tracing.

2. BACKGROUND AND PROBLEM

2.1 Trusted Computing Concepts

The Trusted Computing Group (TCG), an industrial initiative towards the realization of Trusted Computing, has specified security extensions for commodity computing platforms. The core TCG specification is the *Trusted Platform Module* (TPM) [35], a hardware security module embedded in computer mainboards. The TPM provides some cryptographic functions and protected storage for small data such as cryptographic keys.

The TPM supports a trusted boot process by allowing to record measurements of the hardware configuration and software stack during the boot process. Measurements are taken at load-time of the software components (using cryptographic hashing of binaries). These measurements are securely stored in specific TPM registers called *Platform Configuration Registers* (PCRs). Based on these PCR values, the TPM provides the *sealing* functionality, i.e., binding encrypted data to the recorded configuration, and *attestation*, i.e., reporting the system state to a (remote) party by presenting the PCR values digitally signed by the TPM.

2.2 Return-Oriented Programming Attacks

Runtime attacks can change the process behavior and they are not reflected in the attestation of load-time integrity measurement. A traditional vulnerability of programs is the buffer overflow on the stack [2]. An attacker overwrites the saved return address of a function on the stack such that it points to injected code, which the attacker also places on the stack. Subsequent instructions are executed on the stack, which is now interpreted as code and not data. Marking the stack as non-executable [24] prevents such code injection attacks. Modern processors from AMD and Intel provide a non-executable bit for each memory page, which operating systems can set for data pages.

In contrast, in return-oriented programming (ROP) attacks [5, 31, 32], the attacker overwrites the stack with addresses that point to already existing code. Addresses that are mainly used point to the standard C library (libc) because it is linked into most Unix programs. But other libraries or parts of the program may serve as target addresses as well.

Instead of calling a function, ROP carefully sets return addresses on the stack to the middle of instruction sequences (gadgets) that end with a return instruction. After executing the instructions of the gadget, the return takes the next address from the stack where execution continues, and increments the stack pointer. Hence, by carefully crafting the data that overwrites the stack, an attacker can point to existing code sequences. The stack pointer effectively determines the program control flow then and acts like an instruction pointer. This attack completely circumvents protection methods based on non-executable data pages.

Buchanan et al. [5] have shown that this attack can be generalized from x86 to RISC processor architectures. They even showed that it is possible to build a high-level language and a compiler based on those gadgets, which enables attackers to easily construct the stack to include return addresses which result in the execution of arbitrary behavior within a vulnerable process.

The authors of [5] briefly mention some strategies to be worth to explore as possible mitigating techniques for ROP attacks. However, they state that "any 'Trusted Computing' technology using cryptographic attestation" cannot detect ROP attacks. The reason is that such attestation covers only load-time integrity measurements. Therefore, we want to extend the TCG based integrity measurement and attestation techniques to include runtime checks. We aim at proving that trusted computing, with certain extensions, can be adopted to detect return-oriented programming attacks.

3. ARCHITECTURE FOR DYNAMIC INTEGRITY MEASUREMENT

To explain the main concept of our dynamic integrity measurement architecture (DynIMA), we consider a simple process model. A process consists of code and data. Code is the static part of the program binary, whereas the data contains runtime variables and, in particular, the stack, which holds function arguments and return addresses of function calls.

The general idea of DynIMA is to combine load-time integrity measurement with dynamic tracking techniques. A program's code will be instrumented with tracking code that will perform integrity-related runtime checks. To realize this idea, we propose to change the usual program loader of the operating system. It should not only include static integrity measurement facility (as in IMA [29]), but also a new component that rewrites the code of programs to be loaded to include special tracking code that monitors dynamic events of the program and maintains tracking data. Tracking code will be generically instrumented to program binaries because we aim to track common patterns of ROP attacks and not program-specific behavior. Hence, we do not require the source code of programs to be monitored by DynIMA.

Program loading.

The sequence of loading a program in DynIMA works as follows: When a program is going to be loaded, the loader measures the static binary of the program and a tracking instrumentation component rewrites the code to include tracking code. The process of the program is then started and the tracking code stores tracking data in the data segment. The tracking code will dynamically update tracking data as tracking events occur. The tracking events are pre-defined for all program binaries.[1] For instance, based on the measurement of the binary, a different event pattern and tracking code could be applied. Which pattern and tracking code

[1]DynIMA could be extended that tracking events also contain individual, program-specific events. However, this would require knowledge on details of each program and probably require the source code. This is out of scope of this paper since we aim to detect ROP attacks, which are a general problem.

has to be applied is defined in a tracking policy of DynIMA. The components of the loader and the tracking instrumentation are contained within the Process Integrity Manager (PIM). In contrast to a program monitor such as [12], PIM instruments the program (and other parts of the operating system) with tracking code, which results in a distributed monitor. Parts of the monitoring are performed in PIM and the operating system, and parts are performed within the program's process. Figure 1 shows the DynIMA architecture.

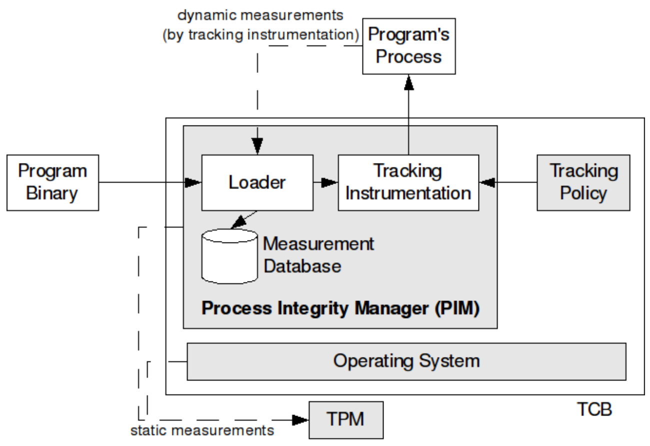

Figure 1: Architecture of DynIMA

Attestation.

We assume the TCB (including, BIOS, bootloader, operating system, and PIM) to be measured statically at boot time and recorded in the PCRs of the TPM. Any other program is loaded and measured by the PIM, similar to static measurement as done in IMA [29] (in fact we use IMA for the static part). The result of the static measurement and the dynamic tracking is stored on a per-process basis within the PIM. Hence, an attestation procedure is composed of the following steps:

1. A verifier requests attestation of a certain program.
2. PIM requests a PCR quote from the TPM to attest to the TCB (including the PIM).
3. PIM digitally signs the measurement and current tracking data of the program which attestation is requested.
4. PIM sends the quoted PCRs (signed by the TPM) and the dynamic measurement of the program (signed by PIM) to the verifier.
5. The verifier verifies the presented data and decides further steps according to the result.

4. TRACKING INSTRUMENTATION

In this section we briefly explore two possible code instrumentation techniques that could be applied as our tracking instrumentation in DynIMA. The integration and implementation of these techniques in our architecture is currently work-in-progress.

4.1 Using Taint Tracking

As one possible countermeasure to prevent attacks that are based on buffer overflows and return-oriented programming we consider *dynamic taint analysis*. Dynamic taint analysis marks any untrusted data as tainted, tracks the propagation of tainted data during program execution, and alerts or terminates the program if tainted data is misused. Misuse of tainted data is, for instance, using tainted data as a pointer, since most buffer overflow attacks are based on changing return addresses and function pointers.

To design a dynamic integrity measurement architecture, we propose a combination of trusted computing and taint tracking to achieve both load-time and run-time integrity. The design we propose is depicted in Figure 2.

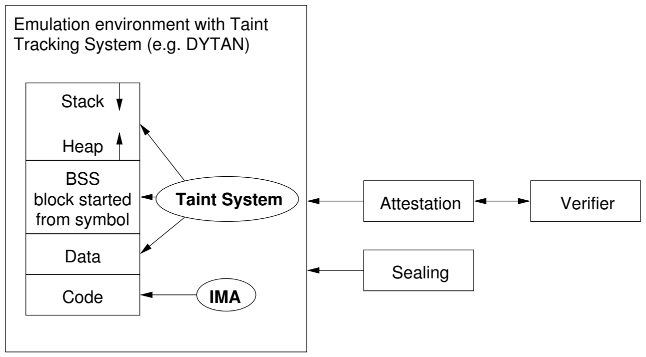
Emulation environment with Taint Tracking System (e.g. DYTAN)
Stack
Heap
BSS block started from symbol
Data
Code
Taint System
IMA
Attestation
Verifier
Sealing

Figure 2: DynIMA with taint tracking

On the left side of Figure 2, we see a usual Unix process that is running in an emulation environment that contains a taint tracking system as plug-in. Only the Code (or Text) segment is protected by IMA at load-time. In addition, dynamic taint analysis monitors data flow integrity. The static results from IMA and the dynamic results from taint tracking are both collected by PIM. If the taint tracking detects an integrity violation, the program will be marked as tainted and PIM records this accordingly. Hence, a verifier can check through dynamic attestation (see above) whether the system has been compromised at run-time. Moreover, the verifier must be able to verify that the taint tracking system is running on the verified system for the application he wants to verify. Besides, the taint tracking system should be based on binary instrumentation. Otherwise, some libraries have to be excluded from the taint analysis since the source code of libraries is not always available or they are written in assembly language.

Tools that enforce dynamic taint analysis are usually implemented in Binary Instrumentation Frameworks (DBIs). Therefore we implemented a tool that tries to detect ROP attacks by counting any instruction between two return instructions in the PIN framework [20]. Since instruction sequences used in ROP attacks range from two to five instructions [31], we increment a counter if the instructions between two returns are less or equal than five. Further, because the small instruction sequences are all chained consecutively together to form a gadget, our pintool reports a ROP attack if three of these small instruction sequences were executed one after another. As a first test, we deployed a test program that exploits a buffer overflow vulnerability in a program and runs a ROP attack. Our pintool was able to detect the attack because of the small instruction sequences executed in the ROP attack. These are promising results, and in future work we have to identify the appropriate parameters for detecting return-oriented programming in general in order to distinguish a gadget call from ordinary short instruction sequences.

4.2 Using Dynamic Tracing

While taint tracking requires to instrument the program before execution, dynamic tracing techniques exist that allow to instrument code on-the-fly. DTrace [6] is a dynamic instrumentation framework for tracing kernel and processes operation in production systems. It uses code rewriting techniques to dynamically instrument both pre-defined probe points in the kernel but also arbitrary user process instructions. If probes are not activated, there is no tracing code instrumented in the processes or kernel. This allows dynamic instrumentation of running systems without restarting the processes under observation.

Using a dynamic tracing mechanism such as DTrace would improve the flexibility of our design since tracing can be switched on and off as needed. Moreover, we expect a performance gain because when tracing is switched off, no tracing code will be actually executed.

DTrace was also examined how it can be used for reverse engineering tasks [34]. To overcome the constraints of the D language provided by DTrace, the authors use a combination of DTrace and the object-oriented programming language Ruby. The framework they propose is called RE:DTrace and is distributed with scripts that are able to detect stack and heap based buffer overflow attacks. We have to investigate if it is possible to use their work and their ideas to extend RE:DTrace in a way to detect ROP attacks.

We are currently investigating DTrace as another means to detect ROP attacks. Therefore, we instrument the libc library and try to detect whether functions are called from the beginning or in the middle of instruction sequence. The latter would indicate a gadget call of a ROP attack.

To track all entries and returns to and from libc, we developed the following script in the D language:

```
1. pid$target:libc::entry {activate[probefunc]++;}

2. pid$target:libc::return /activate[probefunc]==0/
   {printf("Return of %s without entry!",probefunc);}

3. pid$target:libc::return /activate[probefunc]!=0/
   {activate[probefunc]--;}
```

The array `activate[probefunc]` is initialized with null and holds for every function in libc a counter value, whereas a value greater than null means that the function is currently being executed. Therefore the first rule increments the counter in the case the function is called. The third rule decrements the counter if the function returns ordinary to its caller. However, the probes for the second rule will fire before rule three is applied if the counter value is set to null, which means that DTrace has not registered an entry to this function although a return from this function is issued. This could be an indication for a gadget call in a ROP attack. As we enabled these probes for some ordinary programs we found out that there are some functions (e.g. memmove and memcopy) were the second rule fires although no ROP attack occurred. Our ongoing work includes separating these functions in order to reduce the number of false positives.

5. RELATED WORK

In the literature, there are several works regarding load-time integrity checks. For example, secure boot mechanisms [3] and trusted computing enhanced Linux kernel modules like Enforcer [21] perform integrity checks of the loaded operating system and other modules. The Integrity Measurement Architecture (IMA) [29] for Linux uses a TPM to record measurements of any dynamic executable content at load-time from the BIOS all the way up to the application layer. While the TCG attestation method is based on hashes of program binaries, several improvements have been proposed in the literature to overcome some inherent drawbacks, such as lack of flexibility for program updates or privacy concerns on reporting the exact platform configuration [26, 28, 15, 17, 21]. They mainly use certificates or abstract properties instead of binary hashes. However, all those approaches are still based on load-time integrity measurement, and are not capable of detecting (malicious) changes in the measured programs during their runtime.

In addition to load-time measurement, there are also approaches to extend attestation with runtime integrity measurement techniques. Semantic remote attestation [12] exploits security properties of programming languages, e.g., type-safety, which allow for certain properties of runtime attestation. However, it requires a trusted language-specific environment. In contrast, we target runtime attestation of arbitrary program binaries independent of their programming language. PRIMA [14] extends simple load-time integrity measurement with information flow integrity. High integrity processes have to have an input filtering interface, or they are not allowed to access low integrity data. However, to the best of our knowledge PRIMA is not able to prevent ROP attacks. Runtime monitors for the Linux kernel have been proposed [19, 25] that analyze and inspect dynamic structures of the kernel memory. They periodically check function pointers [25], or verify dynamic data structures [19], which is also performed in response to system events. These approaches are useful to protect the operating system kernel, but do not target ROP attacks on user programs. Hence, they are complementary to our work.

The closest work to our architecture is ReDAS [16], a similar framework for dynamic attestation with runtime integrity monitoring of programs. Their approach differs to ours in two main aspects: (i) ReDAS requires the source code of each program to perform an analysis of dynamic properties before execution, and (ii) ReDAS cannot detect return-oriented programming attacks in general because it monitors applications on the granularity of system calls. But ROP attacks can change program behavior without any function calls by exploiting code fragments in the middle of functions [5]. In contrast, we aim to detect any ROP attack by monitoring ROP-specific dynamic behavior, which is generic for all programs, and we do not require the source code of applications.

Several architectures and tools for dynamic taint analysis were proposed and developed in the past years. On the one hand, software-based systems were deployed, whereas some are based on binary instrumentation [8, 9, 23, 27] and others are compiler-based [1, 7, 18, 36]. On the other hand, several hardware-based solutions were presented [10, 11, 33] that reduce overhead associated with software-based solutions, but require hardware extensions. We use these techniques, in particular binary instrumentation, in DynIMA in order to detect ROP attacks.

The authors of [4] describe in their position paper, which inspired much of our work, how tracing techniques such as DTrace could be used to complement load-time integrity measurement with runtime policy enforcement. In their vision software developers should define application-specific events of their programs as policy. In contrast, we aim at defining common events that are tracked for all programs in order to detect runtime changes, namely return-oriented programming attacks.

6. CONCLUSIONS AND FUTURE WORK

We have described how trusted computing concepts can be extended with tracking instrumentation in order to bridge the gap between static load-time and dynamic runtime integrity measurement. The DynIMA framework we propose is to the best of our knowledge the first framework that provides both load-time and runtime integrity for program binaries without knowledge of their source code and even under the presence of attacks that are based on return-oriented programming.

Future work has do address some open questions, e.g., the design of the operating system loader: Should every program run with PIM or is it possible to exclude some programs from instrumentation to achieve better performance? We have to take into consideration that instrumentation, especially in the case of taint analysis, can highly impact the performance of the overall system. Further, we have to find meaningful values to integrate in the existing integrity measurement to include the tracking results in remote attestation procedures. Moreover, should we measure the program binaries *before* or *after* applying the code rewriting that includes the dynamic tracking? Does it have any meaningful implications in doing either way? We believe not if we assume the code rewriting as a trusted and deterministic process. However, on different platforms different code rewriting implementations could result in different measurements afterwards, resulting in unknown values to a verifier during attestation. We also have to take into consideration that code rewriting changes the process image and can hinder the execution of evaluated and certified programs. One approach to overcome this obstacle is to certify the rewriting techniques. The combination of the certified program and the certified rewriting technique then allows usual execution of the program. We will explore these and other issues in our ongoing work to complete DynIMA to fill the gap between load-time and runtime integrity measurement.

7. REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *CCS '05: Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353. ACM, 2005.

[2] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), 1996.

[3] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 65–71, Oakland, CA, May 1997. IEEE Computer Society.

[4] S. Bratus, M. E. Locasto, A. Ramaswamy, and S. W. Smith. New directions for hardware-assisted trusted computing policies. In *Conference on the Future of Trust in Computing (FTC 2008)*, June 2008. `http://www.cs.dartmouth.edu/~sws/pubs/berlin.pdf`.

[5] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.

[6] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of USENIX 2004 Annual Technical Conference*, pages 15–28, Berkeley, CA, USA, 2004. USENIX Association.

[7] W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *CCS '08: Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 39–50. ACM, 2008.

[8] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC 2006)*, pages 749–754. IEEE, 2006.

[9] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing*, pages 196–206, 2007.

[10] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th International Symposium on Microarchitecture*, pages 221–232, 2004.

[11] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 482–493, 2007.

[12] V. Haldar, D. Chandra, and M. Franz. Semantic remote attestation: A virtual machine directed approach to trusted computing. In *USENIX Virtual Machine Research and Technology Symposium*, 2004.

[13] Internet Crime Complaint Center. 2008 Internet Crime Report. `http://www.ic3.gov/media/annualreport/2008_IC3Report.pdf`, 2008.

[14] T. Jaeger, R. Sailer, and U. Shankar. Prima: policy-reduced integrity measurement architecture. In *SACMAT '06: Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 19–28, New York, NY, USA, 2006. ACM.

[15] S. Jiang, S. Smith, and K. Minami. Securing web servers against insider attack. In *17th Annual Computer Security Applications Conference (ACSAC)*, pages 265–276, 2001.

[16] C. Kil, E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang. Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. In *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009)*, 2009. to appear.

[17] U. Kühn, M. Selhorst, and C. Stüble. Realizing

property-based attestation and sealing with commonly available hard- and software. In *STC '07: Proceedings of the 2nd ACM Workshop on Scalable Trusted Computing*, pages 50–57. ACM Press, 2007.

[18] L. C. Lam and T.-C. Chiueh. A general dynamic information flow tracking framework for security applications. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 463–472. IEEE Computer Society, 2006.

[19] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, and C. D. McDonell. Linux kernel integrity measurement using contextual inspection. In *Proceedings of the 2nd ACM Workshop on Scalable Trusted Computing (STC'07)*, pages 21–29. ACM, 2007.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, volume 40, pages 190–200, New York, NY, USA, June 2005. ACM Press.

[21] R. Macdonald, S. Smith, J. Marchesini, and O. Wild. Bear: An open-source virtual secure coprocessor based on TCPA. Technical Report TR2003-471, Department of Computer Science, Dartmouth College, 2003.

[22] Microsoft Corporation. Bitlocker drive encryption, July 2007. `http://www.microsoft.com/technet/windowsvista/security/bitlockr.mspx`.

[23] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed Security Symposium*, 2005.

[24] PaX Team. `http://pax.grsecurity.net/`.

[25] N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *CCS '07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 103–115. ACM, 2007.

[26] J. Poritz, M. Schunter, E. Van Herreweghen, and M. Waidner. Property attestation—scalable and privacy-friendly security assessment of peer computers. Technical Report RZ 3548, IBM Research, May 2004.

[27] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM Symposium on Microarchitecture*, pages 135–148, 2006.

[28] A.-R. Sadeghi and C. Stüble. Property-based attestation for computing platforms: Caring about properties, not mechanisms. In *The 2004 New Security Paradigms Workshop*, pages 67–77. ACM, 2004.

[29] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th USENIX Security Symposium*, pages 223–238, 2004.

[30] SANS Institute. SANS Top-20 2007 Security Risks. `http://www.sans.org/top20/2007/top20.pdf`, Nov. 2007.

[31] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS '07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 552–561. ACM, 2007.

[32] Solar Designer. "return-to-libc" attack. Bugtraq, 1997.

[33] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS-XI: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96. ACM, 2004.

[34] Tiller Beauchamp and David Weston. Dtrace: The reverse engineer's unexpected swiss army knife. `http://www.poppopret.org/DTrace-Beauchamp-Weston.pdf`, 2008.

[35] Trusted Computing Group. TPM main specification. Specification Version 1.2 rev. 103, July 2007. `https://www.trustedcomputinggroup.org/specs/TPM/`.

[36] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, pages 121–136, 2006.