Static Analysis Based Invariant Detection for Commodity Operating Systems

Jinpeng Wei, Feng Zhu
School of Computing and Information Sciences
Florida International University
Miami, FL, USA
{weijp, fzhu001}@cs.fiu.edu

Yasushi Shinjo
Department of Computer Science
University of Tsukuba
Tsukuba, Ibaraki, Japan
yas@cs.tsukuba.ac.jp

Abstract—**The recent interest in runtime attestation requires modeling of a program's runtime behavior to formulate its integrity properties. In this paper, we study the possibility of employing static source code analysis to derive integrity models of a commodity operating systems kernel. We develop a precise and static analysis-based *global invariant* detection tool that overcomes several technical challenges: field-sensitivity, array-sensitivity, pointer analysis, and handling of assembly code. We apply our tool to Linux kernel 2.4.32 and identify 141,279 global invariants that are critical to its runtime integrity. Furthermore, comparison with the result of a dynamic invariant detector reveals 17,182 variables that can cause false alarms for the dynamic detector. Our experience suggests that static analysis is a viable option for automated integrity property derivation, and it can have very low false positive rate (1 out of 141,280 in our Linux kernel case study) and very low false negative rate (about 0.013%).**

Keywords-integrity modeling; invariants detection; static analysis; tools

I. Introduction

In a cooperative environment, trust among the participating computer systems is vital to the correct functioning of the entire system. However, the widespread exploitations of software vulnerabilities (e.g., buffer overflows) and security breaches undermine the trustworthiness of computer systems in a collaborate environment and thus may put other participating systems at great risk. Therefore, technologies are needed to gauge the trustworthiness of a running computer in a collaborate environment.

Remote attestation is a promising technique that enables a computer system in a cooperative environment to decide whether a target computer (in the same environment) has integrity, e.g., whether it has the appropriate configuration and hardware/software stack, so it can be trusted. The idea of remote attestation has been widely accepted. For example, the trusted platform modules (TPM) [29] chip has become a standard component on modern computers.

Early remote attestation techniques only ensure that a computer is bootstrapped from trusted hardware and software (e.g., operating systems and libraries), but there has been a consensus in recent years that such *static* attestation is not enough [18, 20]. This is because *runtime attacks* such as buffer overflow attacks can invalidate the result of static attestation during the execution of the target system, so a remote challenger cannot gain high confidence in a target system even if it is statically attested [18]. In order to regain high confidence, we must enhance traditional remote attestation with *runtime attestation*, or runtime integrity checking.

One of the fundamental challenges for runtime attestation is the attestation criteria, i.e., the expected integrity properties, of the target system. Other than a few static program states (e.g., code segments and constant data), most of the runtime state space of a system (normal variables, stack, and heap) cannot be trivially characterized. This uncertainty about the criteria results in two classic attestation errors: false positives and false negatives. False positives happen when the remote challenger endorses an overly stringent criterion that a normal (uncompromised) system fails to meet; and false negatives happen when the challenger endorses an overly loose criterion that a compromised system can also meet (i.e., the remote challenger ends up trusting a corrupted computer). It is obvious that both kinds of errors are undesirable for remote attestation.

The root cause for the attestation errors discussed above is the lack of *precise* specifications of expected integrity properties. While under-specification can reduce the rate of false positives by lowering the bar for a target system, it allows a compromised system to obtain trust. On the other hand, over-specification ensures that no compromised system can pass the integrity check, but it may also raise too many false alarms.

Since the integrity properties are attributes of the target system, a precise specification demands a thorough analysis of the target system. Several kinds of approaches have been taken to analyze a target system for its integrity properties. Manual analysis relies on domain expertise to specify and prove the correctness of integrity properties. It is applicable to well-known properties such as the immutability of the Interrupt Descriptor Table (IDT), but it is not scalable to complex software such as the Linux kernel. Therefore, automated tools are much needed to assist a human expert. Dynamic analysis tools such as Gibraltar [7] and ReDAS [18] infer likely integrity properties (called *invariants*) of a system by reading the runtime states (e.g., memory snapshots that contain program variables) of the target system and hypothesizing whether some variables satisfy predefined invariant relationships. One example relationship is that a variable *v* must always have a constant value *k* at runtime. However, a well-known drawback of dynamic analysis is its inability to explore all possible program execution paths. As a result, dynamic analysis may generate false invariants. For example, Gibraltar generates about 4,673 false invariants for Linux kernel 2.4.20 [7]. A typical solution to overcome such shortcomings is to use a large set of test cases. For example, ReDAS created 70 training scenarios and 13,000 training sessions for the *ghttpd* server. However, how to systematically generate a large number of test cases that can trigger all execution paths in a program remains a challenging research problem in general.

In this paper, we explore the application of static analysis for finding integrity properties. The basic idea is to use compiler technology to analyze the behavior of a program to derive its integrity properties, without actually running the program. Static analysis can overcome the limitations of dynamic analysis by exploring all execution paths. For example, if `v=v+2` is found in the *true* or *false* branch of a conditional statement in the target program, then the property that "variable *v* always has a constant value at runtime" is likely *false*. However, a dynamic analysis tool will not be able to observe this assignment if the test cases do not satisfy the condition for the assignment; as a result, a dynamic analysis tool may conclude that *v* is an invariant. Since static analysis has the source code of the program, it has the advantage to reveal all conditions for assignments to a variable, so it can be more precise.

Specifically, we focus on the static detection of one class of integrity properties called *global invariants*. Global invariants are code or data that has constant value at runtime. They can represent critical system integrity properties such as the immutability of the Interrupt Descriptor Table (IDT) and the system call table. Therefore, they have been checked by state-of-the-art integrity monitors [7, 18].

Our first contribution is a program analysis tool that can automatically derive global invariants from source code, using static analysis. Our tool applies compiler technology to analyze the control and data flows (e.g., assignments and function calls) of a target program and reason about the global variables that are invariants. In developing this tool, we have overcome several challenges in large-scale C program analysis, such as field-sensitivity, array-sensitivity, pointer analysis, and handling of assembly code.

Our second contribution is a thorough study of global invariants detection for the Linux kernel using *static analysis*. To the best of our knowledge, there has not been a similar study. Linux kernel is a very complex piece of software posing great challenges for static analysis by its wide use of pointers and complex structures. Our tool is able to process 400,492 lines of Linux kernel (version 2.4.32) code and identify 141,279 global invariants essential to the Linux kernel's runtime integrity. More importantly, by comparing with the results of a dynamic invariant analyzer, we find 17,182 variables that can cause false alarms for the dynamic analyzer, while our static tool only misses 18 true invariants (with false negative rate 0.013%). We also develop an invariant monitor based on the result of the static analysis and the runtime evaluation of the monitor finds *only one* false invariant. Our experience suggests that static analysis is a viable option for automated integrity property derivation, and it can potentially have very low false positive and false negative rates.

The rest of the paper is organized as follows. Section II discusses the modeling of global invariants as an important class of integrity properties. Section III presents an automated global invariants detection tool based on static analysis of source code. Section IV discusses a thorough evaluation of our invariant detection tool by applying it to the Linux kernel. Section V discusses related work, and Section VI concludes the paper.

II. BACKGROUND ON GLOBAL INVARIANTS

In this section, we first discuss the basics of integrity measurement and our assumptions; then we formally define *global invariants* as a class of integrity property.

A. *Background on Integrity Measurement and Security Assumptions*

An integrity measurement system typically consists of three components: the target system, the measurement agent, and the decision maker [20]. Our first assumption is that the measurement agent is isolated from and independent of the target system, therefore it has a true view of the internal states (including code and data) of the target system. This is a realistic assumption due to the popularity of virtual machine monitors [9] and machine emulators such as QEMU [10], and it has also been shown that the measurement agent can run on dedicated hardware such as a PCI card [22]. Our second assumption is that measurement results are securely stored and transferred to the decision maker. This can be supported by hardware such as a Trusted Platform Module (TPM) [29]. The third assumption is that the target system's states (e.g., code and data) may be compromised by a powerful adversary who can make arbitrary modifications; therefore the decision maker can rely on very few assumptions about the trustworthiness of the target system.

Based on these assumptions, the decision maker is given a true view of the target system, and its task is to estimate the "healthiness" of the target system. The healthiness include functional correctness (e.g., a function that is supposed to reduce the priority level of a task is not modified to actually increase the priority level), and non-functional correctness (e.g., the priority level can be modified by a privileged user instead of a normal user). In the following subsections, we model the healthiness as integrity properties.

Moreover, the healthiness of the target system may change over time, because it may be under constant attacks. Therefore, the integrity of the target system may need to be periodically reevaluated.

B. *Definition of Global Invariants*

In theory, any software system can be modeled as an automaton with states and state transitions. For simplicity of presentation, we assume that the system can be in one of n possible states: $s_1, s_{2,} ..., s_n$. Example states are initialization, entering a function, returning from a function, system termination, and so on. And each state is characterized by a particular combination of values of the system's internal variables. Based on this general formalization, we can model runtime software integrity as a set of properties $\{P_1(s), P_2(s), ..., P_m(s)\}$. A runtime property $P_i(s)$ is a function on state s that evaluates to *true* or *false*. If a system state s satisfies all P_i 's, we can say that s is a "healthy" state. Different runtime properties may have different structures, but each of them can be generalized to be a Boolean expression with the operators $\wedge$(and), $\vee$(or), and $\neg$(not). More complex properties can be constructed out of primitive properties using the operators mentioned above. A primitive property has the form $func(v_1(s), v_2(s), ..., v_l(s))$ which takes variables $v_1(s), v_2(s), ..., v_l(s)$ and returns *ture* or *false* ($v(s)$ is the value of v in state s). *func* can have arithmetic operations inside as well as relationship operations such as ==, $\neq$, <, and >.

One special class of primitive property has the form: $v(t) == k, t \in [s_1, s_2)$, where s_1 = "system initialized" and s_2 = "system shutting down". I.e., it stipulates that the value of variable v must be a *known-good* value k during the runtime of the system (assuming that there is a sequence of state transitions from s_1 to s_2). We call such a primitive property a *global invariant*.

C. Relevance in Integrity Protection

Global invariants represent an important class of integrity properties. They may include critical internal control data of the system (e.g., function addresses) that are supposed to remain constant. Examples of such invariants include the Interrupt Descriptor Table (IDT). Another type of global invariants hold security policy data, and the violation of such invariants can directly defeat the corresponding security measures. For example, by tampering with the list of "bad" IP addresses, the attacker can defeat a blacklist-based IDS.

Because of the importance of global invariants to integrity properties, they have been the popular targets for attack by rootkits such as SucKIT, Hacker defender, Turtle rootkit, enyelkm-1.3, Phalanx, AFX, NTIllusion, HE4Hook, and Vanquish. Common examples of such attacked invariants include system call table, System Service Descriptor Table, SYSENTER handler function, Interrupt Descriptor Table (IDT), NDIS handlers, and Interrupt Request Packet (IRP) handlers. It is interesting to note that over the years, the list of such invariants grows as the rootkits attempted to evade rootkit detectors that tried to catch up. The general trend of such growth is towards more sophistication and stealth [8]. On the other hand, global invariants are the basis for many rootkit detection systems such as ReDAS [18], Copilot [22], Livewire [14], and several commercial tools (e.g., [1, 2, 3, and 31]). The fact that such invariants are not supposed to change makes it easier to check their integrity; for example, many rootkit detectors use a clean copy or hash value of a global invariant variable as the baseline to tell whether it has been tampered with by a rootkit.

D. Existing Solutions

Several kinds of approaches have been taken to analyze a target system for global invariants. Manual analysis relies on domain expertise to specify and prove the correctness of integrity properties. It is applicable to well-known properties such as the immutability of the Interrupt Descriptor Table (IDT), but it is not sustainable to counter novel attacks that move their targets to less-known places such as device driver jump tables. Eventually, manual analysis will reach a point where a human expert has difficulty understanding the logic of a system, which calls for automated tools to assist a human expert. Dynamic analysis tools such as Gibraltar [7] and ReDAS [18] infer likely global invariants of a system by reading the runtime states (e.g., memory snapshots that contain program variables) of the target system and hypothesizing whether some variables have constant values at runtime. One well-known drawback of dynamic analysis is its inability to explore all possible program execution paths. As a result, dynamic analysis may generate false invariants. Typical solution to overcome such shortcomings is to use a large set of test cases. For example, ReDAS [18] created 70 training scenarios and 13,000 training sessions for the *ghttpd* server. However, how to systematically generate a large number of test cases that can trigger all execution paths in a program is a challenging and open research problem by itself.

III. AUTOMATED INFERENCE OF GLOBAL INVARIANTS THROUGH STATIC ANALYSIS

A. Overview

We have developed a static analysis-based system to detect global invariants for commodity operating systems kernels. Fig. 1 shows the overall architecture. We apply compiler technology to automatically analyze the control and data flows of a target kernel, e.g., assignments and function calls, to reason about the global variables that are invariants. Assignment recognition supports or rejects the hypothesis that a variable is an invariant, e.g., if a variable is assigned multiple times with different values, it is unlikely a global invariant.

Our system recognizes two kinds of assignments: direct assignment and indirect assignment. Direct assignment recognition is straightforward. Indirect assignment is mainly made through *pointers* in a modern kernel implemented in the

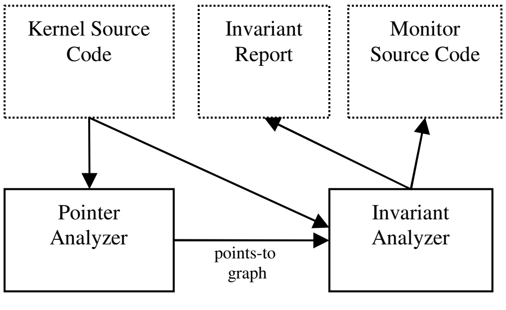

Figure 1. Invariant analysis architecture

C language. In order to recognize indirect assignment through pointers, our system has a pointer analysis component, as shown in Fig. 1.

Our system accepts the merged kernel source code as input, which is fed into the Pointer Analyzer and the Invariant Analyzer. The Pointer Analyzer processes the kernel code and generates the points-to graph, which records the points-to set (i.e., a set of variables) of any pointer variable. The Invariant Analyzer scans the kernel source code including variable declarations and kernel functions. Its major task is to recognize assignment statements, either direct ones or indirect ones. When it scans an indirect assignment statement such as `*p=v`, it queries the points-to graph generated by the Pointer Analyzer and gets the set of kernel variables that can be pointed to by `p`, and notes that all such variables are assigned once by this statement. Internally, the Invariant Analyzer has two functional components: assignment recognition (Section III.C) and invariant recognition (Section III.D).

The output of our system is a report of invariant classification for all global variables in the input kernel. For variables that are not considered invariant, we also report the reason for that decision, e.g., the related assignment statement(s). We have made a sample of such a report available on our web site [34]. Another output of our system is the source code for an Invariant Monitor that can be installed in the analyzed system as a kernel module to monitor the invariants detected. More details of the Invariant Monitor are presented in Section IV.C.1.

B. Design Goals

The major goal for our invariant analysis is high precision, i.e., how to minimize false positive rate and false negative rate. As we discussed in Section I, both kinds of errors are undesirable for remote attestation. Below we discuss the technical challenges in both cases and our solutions, under the context of static analysis of C like programming languages.

The major reasons for false negatives are a lack of fine-granularity and imprecise pointer analysis. If the static analyzer is field-insensitive, e.g., it cannot differentiate individual fields in a C structure, it will regard an assignment to any field of a structure as an assignment to the entire structure; thus the entire structure may become a non-invariant. This means that even if some fields of that structure are invariant and hold critical data such as function pointers, they cannot be protected. This lack of precision obviously causes false negatives. Similarly, lack of support for array sensitivity, i.e., being unable to differentiate individual elements in an array, is another cause for false negatives. From our experience, in a modern kernel such as the Linux kernel, the majority of global data is within some structure or array, which means that a static analyzer that is field and array-insensitive is almost useless. Therefore, our invariant analyzer must be field and array-sensitive (Sections III.C.1 and III.C.2). Another cause of false negatives is the conservativeness of pointer analysis algorithms. If the pointer analysis algorithm is too conservative, e.g., a pointer can point to all global variables, the invariant analyzer would recognize many bogus (or impossible) assignments, and consider a global invariant as a non-invariant as a result. Therefore, we need to develop a precise pointer analysis algorithm. We employ a precise points-to algorithm in our design (Section III.C.3).

The major reason for false positives (i.e., fake invariants) is a failure to recognize legitimate assignments. This can be caused by two reasons: implicit assignments and incomplete points-to analysis. One kind of implicit assignment is assignment by assembly code: since our static analyzer does not understand assembly code, it cannot capture such assignments. Another example of implicit assignment is structure-level assignment: if variables `foo` and `bar` are defined as `struct{int a;int b}foo,bar;` then the assignment `foo = bar` implicitly modifies both `foo.a` and `foo.b`. Another cause of missing assignment recognition is related to the precision of points-to analysis: if it returns an incomplete points-to set for a pointer, the analyzer may miss legal but indirect updates to some variables through that pointer; as a result, the analyzer may mistakenly classify those variables as invariants. In order to capture implicit assignments, we apply heuristics in our analyzer (Sections III.C.5, III.C.4, and III.C.1). In order to avoid incomplete pointer analysis, we employ a precise points-to analysis algorithm (Section III.C.3).

C. Major Design Points of the Assignment Recognition

One component of our invariant analyzer identifies assignments to variables. For programs written in C, modifications to a variable occur in two forms: direct assignment and indirect assignment. In the former case, the said variable is the left hand side of an assignment statement (e.g., `v` in `v=k+3`). In the latter, the said variable is assigned indirectly through a pointer that references it (e.g., `p=&v,...,*p=k+3`). In order to capture the second case, the detector needs to first find out the points-to set of the pointer (via a points-to analysis [5]), and then note that each target in the points-to set is assigned indirectly.

In the rest of this section, we discuss how our design satisfies our goals outlined in Section III.B.

1) Field Sensitivity

To achieve the desired field sensitivity, our analyzer uses lexical names to disambiguate structure field references (e.g., `p->a` is considered a different memory location from `p->b`), in a way similar to Wagner [30]. This enables our analyzer to capture explicit assignments to structure fields. Moreover, our analyzer treats structure level assignments as implicit assignments to the individual fields. For example, if variables `foo` and `bar` are defined as `struct{int a;int b}foo, bar;` then the assignment `foo = bar;` is translated into `foo.a = bar.a; foo.b = bar.b`.

2) Array Sensitivity

Array sensitivity is another method for our analyzer to achieve fine-granularity. The basic idea is to treat each element of an array as an independent variable. For example, the array `int d[3]` is treated as three variables `d[0]`, `d[1]`, and `d[2]`. Our analyzer can handle arrays of arbitrary dimension. Finally, our analyzer can recognize pointers into arrays. For example, if the analyzer sees `int *p = d;` it can interpret `*(p + 1)` as the same as `d[1]`.

3) Pointer Analysis

Our invariant analyzer performs points-to analysis in order to recognize indirect assignments through pointers. Based on the analysis in Section III.B, we know that the accuracy of the points-to analysis algorithm is the key for reducing false positives and false negatives. Therefore, our pointer analyzer is based on the generalized one level flow (GOLF) algorithm [11], which is among the most precise pointer analysis algorithms, achieving precision close to Anderson's algorithm [5]. Our pointer analyzer is built on top of the field-sensitivity (Section III.C.1) and array-sensitivity (Section III.C.2) capabilities to return fine-grained points-to targets. For example, it would return individual structure field `foo.b` instead of the entire structure `foo`. Finally, it can also contribute to field-sensitivity and array-sensitivity. For example, in `struct{int a;int b}bar, *p`, if `p`'s points-to set includes `bar`, then an assignment to `p->a` is considered an indirect assignment to `bar.a`.

4) Union Support

Our analyzer also supports unions: each field of a union is treated as an alias of other fields in the same union. This means that an explicit assignment to one field of a union is an implicit assignment to all the other fields. Therefore, if one field of a union is not an invariant, other fields of the union are not invariants, either.

For example, in `union uarg{int a; int b}c`, `c.a` and `c.b` are treated as different variables; if `c.a` is not an invariant, `c.b` is not an invariant, either.

5) Heuristics-base Assignment Recognition

The use of assembly code in the kernel poses difficulties to our static analysis. Because our analyzer only recognizes C code, variable reads or writes by assembly code are not visible to it. One prominent example is `get_current()`, which returns a pointer to the task structure of the current process. Because this function uses assembly code, several chains of pointer dependency are broken, and our static analysis suffers inaccuracy as a result. To overcome these inaccuracies caused by assembly code, we apply a *function prototype-based heuristic*. The basic idea is to summarize the effect (in terms of assignments to the input parameters) of assembly code inside a function body to bridge the "analysis gap". For example, the function `memcpy`() copies a block of memory to another block of memory, so it can change the target memory and thus should be treated as a kind of implicit assignment. This list of functions includes `copy_from_user`, `memset`, `memcpy`, `spin_lock`, `read_lock`, `write_lock`, `down`, `up`, `clear_bit`, `set_bit` and their variants. We identify this kind of functions in two steps: first, our static analysis reports all functions that contain assembly code in their bodies; second, we manually analyze the reported functions to see if any assignment is performed in the assembly code. For function `get_current`(),we assume it can return a pointer to the global variable `init_task_union.task`.

D. Invariant Recognition

The second component of our invariant analyzer recognizes global invariants based on the assignments to each variable.

Because the definition of global invariants only concerns the variables' value after system initialization, we treat assignments during system initialization differently than those during normal execution of the system. Specifically, a global invariant can be assigned multiple possible values during initialization, as long as it is not assigned during normal execution. On the other hand, assignments at normal execution time typically indicate that a variable is not an invariant. Being assigned differently during system initialization is quite possible for some global invariants whose known-good values depend on hardware configuration; they can get several different values depending on the hardware features detected during system initialization.

In our design, each global variable is associated with a flag that indicates whether it is an invariant and a *legal value list* that contains its possible values. In the beginning, all global variables are marked as invariants and all legal value lists are empty.

Our invariant analyzer first scans global variable declarations and initialization functions (e.g., those with "`__init`" directives). If a global variable, which is marked as an invariant, is assigned a constant value, the analyzer adds this value into the variable's legal value list. On the other hand, if a global variable is assigned a non-constant value, the analyzer marks it as a non-invariant.

After this scan, if a global variable's legal value list is still empty, the analyzer adds a default value into the list, based on the type of the variable (e.g., `0` for an integer variable).

Next, the invariant analyzer scans the remaining kernel functions. If a global variable, which is marked as an invariant, is assigned a non-constant value, or a constant value but the value is not in its legal value list, the analyzer marks it as a non-invariant.

At the end of the kernel code scanning, our analyzer generates a report about the invariant status of all global variables, based on their flags. For those non-invariants, the report also includes the reason, e.g., the related assignment statement(s) in the kernel source code, for in-depth investigation by a human expert.

E. Implementation

We implement a static invariant detector based on the C Intermediate Language (CIL) [21]. Our pointer analyzer is implemented in 5,000 lines of Ocaml code, and our invariant analyzer is implemented in 3,500 lines of Ocaml code.

IV. Evaluation

In this section, we report a large-scale evaluation of our invariant detection tool, using Linux kernel 2.4.32 as the input kernel. Our evaluation mainly focuses on the precision of the detected invariants. We also briefly report performance of our detection tool at the end.

A. Metrics, Methodology, and Test Cases

We choose two common metrics to evaluate the precision of the detected global invariants for the Linux kernel:

- **False positives** happen when variables that can legally change their values are mistakenly recognized as invariants. A monitor can generate false alarms when such "fake" invariants change their values during normal execution.
- **False negatives** happen when a true invariant is not recognized as such. As a result of false negatives, a monitor may fail to detect rootkits that modify thus violate the true invariants. In other words, a rootkit can evade detection as a result of false negatives.

We measure the false positive and false negative rates of the static invariant detection in two ways: (1) comparing with the result of a dynamic invariant detector, and (2) running against real software (benign or malicious).

Table I shows the set of benign test programs that we ran. Worth noting among all the programs is the Linux Test Project (*ltp* version 2005), which includes more than 700 test cases that test the Linux kernel in many aspects (such as system calls and file system functionality), and more than 60 test cases that exercise the basic functionalities of the network.

We also tried to run rootkits in our test environment. Since the Linux kernel version that we analyzed is relatively old (2.4.32), most of the publicly available rootkits were not able to run on this kernel. In fact, we were able to run only the SucKIT 2 rootkit.

B. Comparing with a Dynamic Invariant Detector

We develop a dynamic invariant detector (as a loadable kernel module, or LKM) that periodically reads the values of the global variables of the kernel during a training phase and finally reports those variables whose values do not change during the training; these variables are the dynamically-detected invariants. In order to increase the accuracy of the result, we run the test programs in Table I to trigger modifications to the global variables.

Table II summarizes the invariant analysis results for the `.data` and the `.rodata` segments of Linux kernel 2.4.32. The second column is the total number of global variables (with field and array-sensitivity). The third column shows the number of statically-detected invariants out of all the variables, and the fourth column shows the number of dynamically-detected invariants out of all the variables.

TABLE I. TEST PROGRAMS USED IN THE EVALUATION

Test program	*Description*
ltp-2005	Linux Test Project: open source test suites that validate the reliability, robustness, and stability of Linux
Iperf [33]	A network testing tool that measures the throughput of a network, thus exercising the network subsystem of the kernel
Andrew benchmark	A file system benchmark
Miscellaneous	Kernel compilation, ssh, scp, common commands

TABLE II. OVERALL RESULT OF THE INVARIANT DETECTION

Segment	*# Variables*	*# Static inv.*	*# Dynamic inv.*
.data	154,132	136,778	153,978
.rodata	4,502	4,502	4,502

TABLE III. COMPARISON OF THE STATIC AND DYNAMIC ANALYSIS RESULTS FOR THE .data SEGMENT (FN: FALSE NEGATIVE; FP: FALSE POSITIVE).

Category	*Total #*	*# Error static*	*# Error dyna.*
S.NI, D.NI	154	0	0
S.NI, D.I	17,200	18(FN)	17,182(FP)
S.I, D.NI	0	0	0
S.I, D.I	136,778	1(FP)	1(FP)

From Table II we can see that both static and dynamic invariant detection achieve 100% accuracy on the `.rodata` segment, which is expected because variables in the `.rodata` segment are supposed to remain constant.

The dynamic invariant detector reports that 99.9% of the variables in the `.data` segment are invariant. This is not very surprising, because our test cases may not be able to trigger every possible update to a global variable, so there may be many false invariants in the dynamically-detected set. Comparatively, static analysis reports 88.7% of the variables in the `.data` segment as invariants.

Table III gives a more in-depth comparison of the results of the static and dynamic analyzers for the `.data` segment. We classify each global variable according to how the two kinds of analyzers think about its invariant status. Since each variable can be considered invariant or non-invariant by each of the analyzer, there are four combinations. For example, the category "S.NI, D.I" includes all variables that are considered non-invariant by the static analyzer but invariant by the dynamic analyzer.

We can see that there are in total 154 variables that both analyzers agree to be non-invariants. We are confident about the correctness of the results because the dynamic analyzer classifies a variable as non-invariant only if it observes that the value of the variable does change at runtime. Therefore, the non-invariants reported by the dynamic analyzer must be truly non-invariants.

Next, we see from Table III that 17,200 variables are classified as non-invariants by the static analyzer but invariants by the dynamic analyzer. Here we cannot trust the dynamic analyzer because it may not observe a legal but conditional assignment due to the incompleteness of the test cases, and we cannot trust the static analyzer, either, because its points-to analysis is conservative.

To find the ground truth about these 17,200 variables, we manually verify whether they are indeed non-invariants. This verification task seems daunting, but it is actually made much easier by the following facts about our static analyzer: (1) if a variable is directly modified, the assignment statement logged in the analysis report is straightforward evidence that the variable is a non-invariant; (2) if a variable is only indirectly modified through a pointer, our analyzer outputs the relevant statements from the source code that support the points-to relationship, which is relatively straightforward to verify by a human (e.g., Fig. 2 is a portion of our analysis report that shows why `ctrl_map[2]` can be indirectly modified through the pointer variable `key_map`); (3) because our analysis is array sensitive, we can generalize from one confirmed non-invariant array element to all other elements in the same array. E.g., given an array `arr` of size 1024, if we confirm that `arr[0]` is a non-invariant due to an assignment to `arr[i]`, then we can conclude that `arr[1]` through `arr[1023]` are all non-invariants. In other words, we can confirm non-invariants in batches, which significantly speed up the verification. Because of the above reasons, it takes one graduate student about 20 hours to finish the verification of these 17,200 variables. As a result, we are able to confirm that 17,182 of such variables are non-invariants, i.e., they can be modified by assignments at runtime. Below we outline some examples:

- `struct kbdiacr accent_table[256]` stores the *accented* symbols (or characters) of the console keyboard and can be changed by an `ioctl` system call with command 0x4B4B, specifically, `copy_from_user(accent_table,a->kbdiacr, ct*sizeof(struct kbdiacr)` in drivers/char/vt.c. However, the dynamic invariant analyzer reports this entire array as invariants because our test cases do not make such an `ioctl` system call.
- The array `static char buf[1024]` in panic.c is used to hold kernel panic messages whenever `panic`() is called. Specifically, it is written into by `vsnprintf(buf, sizeof(buf), fmt, args)`. Therefore, this array is obviously not invariant. However, since our dynamic analysis test cases do not trigger a kernel crash, it cannot see any changes to `buf`; so it mistakenly concludes that the entire array `buf` is invariant.
- Fig. 2 shows how `ctrl_map[2]` can be indirectly modified through a pointer.

Fig. 3 shows the distribution of the two kinds of evidence (direct assignment and indirect assignment) applicable to the 17,336 non-invariants confirmed (including the 154 non-invariants in the "S.NI, D.NI" category). We have merged the non-invariants recognized via heuristics (Section III.C.5) into the direct or indirect group depending on whether the address of the target variable is directly taken. Note that more than one kinds of evidence may be applicable to a variable depending on how the kernel modifies it.

```
<Name>ctrl_map[2]</Name>
<Invariant>No</Invariant>
<Reason1>
*(key_map + 0) = (unsigned short )(((2 << 8) | 126) ^ 61440);
vt.c:224, Indirectly modified through key_map.

Path from ctrl_map[2] to key_map:
<Label>ctrl_map[2]</Label>
<STMT>ctrl_map=&ctrl_map[2]  defkeymap.c:65</STMT>
<Label>l_473154</Label>
<STMT>key_maps[4]=ctrl_map  defkeymap.c:141</STMT>
<Label>l_479876</Label>
<STMT>key_map = key_maps[tmp.kb_table];vt.c:174
</STMT>
```

Figure 2. Snippet of the analysis report that shows why ctrl_map[2] can be indirectly modified through the pointer key_map

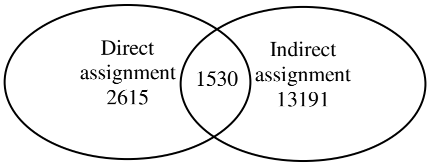

Figure 3. Distribution of evidence applicable to the 17,336 non-invariants confirmed

From Fig. 3 we can see that 76% (13,191 out of 17,336) of the confirmed non-invariants can be modified only indirectly through a pointer. This confirms the wide-scale use of pointers in the Linux kernel to manipulate memory and it also means that for any static invariant detector of the Linux kernel, the points-to analysis part is critical for the overall precision.

We further look at the type and meaning of the 17,182 confirmed non-invariants in the "S.NI, D.I" group to see whether the classification makes sense. We coarsely divide them into several categories, such as list heads, locks, accounting information (e.g., counters), auditing data, resource management information (e.g., page tables and memory zone lists), configuration data, and driver-specific data. Table IV shows example variables for each category.

From this analysis, we feel that the static analysis results make sense. For example, list heads, locks, and performance counters should be dynamic, so they should not be invariants. Unfortunately, our dynamic analyzer classifies this large group of non-invariants in the wrong way, due to the incompleteness of test cases. We believe that these 17,182 non-invariants highlight the relative advantage of static analysis over dynamic analysis.

The 18 false negatives in the "S.NI, D.I" group are the fields of a global structure called `i810_fops` (e.g., `i810_fops.ioctl`, `i810_fops.read`, `i810_fops.write`, etc). For them, our static analyzer does not provide convincing evidence for the points-to relationship and our manual analysis indicates that they should be invariants. These false negatives illustrate the limitation of points-to analysis, which has been shown to be undecidable in general [16]. Given the total number of real invariants (141,297), our static analyzer has a false negative rate of 0.013% (18 out of 141,297).

Continue on Table III, we see that no variable is classified as invariants by the static analyzer but non-invariants by the dynamic analyzer. Such variables, if they exist, would be false positives for the static analyzer. The absence of such variables suggests that our static invariant detector has low false positive rate.

The last row of Table III shows that both analyzers believe that 136,778 kernel variables should be invariants. Since our static analyzer does not provide evidence for invariants, we cannot verify the correctness statically. Similarly, the dynamic analyzer does not provide evidence to prove invariants, either.

TABLE IV. EXAMPLES OF NON-INVARIANTS

Category	*Example variables*
List heads	acpi_bus_drivers.next arp_tbl.gc_timer.list.next console_callback_tq.list.next random_read_wait.task_list.next tcp_tw_timer.list.next
Locks	context_task_wq.lock.lock dev_base_lock.lock exec_domains_lock.lock floppy_usage_lock.lock hash_table_lock.lock
Auditing information	kernel_module.archdata_end kernel_module.archdata_start kernel_module.ex_table_end kernel_module.ex_table_start kernel_module.kallsyms_end kernel_module.kallsyms_start
Accounting information	console_sem.count.counter con_buf_sem.count.counter dev_probe_sem.count.counter init_fs.count.counter init_mm.mm_user.counter
Resource mgmt data	contig_page_data.node_zonelists[0].zones[0] contig_page_data.node_zones[0].free_area[0].map contig_page_data.node_zones[0].nr_cache_pages contig_page_data.node_zones[0].nr_inactive_pages contig_page_data.node_zones[0].nr_active_pages
Configuration data	FDC2,FLOPPY_DMA,FLOPPY_IRQ, can_use_virtual_dma,fifo_depth
Driver-specific data	eth0_dev.allmulti, eth0_dev.dev_addr[0] eth0_dev.tx_queue_len, eth1_dev.change_mtu, eth1.base_addr, eth1.broadcast[0]

Therefore, we decide to experimentally verify the invariant results.

C. Experimental Evaluation

1) Implementation of the Invariant Monitor

We implement an Invariant Monitor (in the form of a LKM) that periodically checks the 141,280 statically detected invariants in the memory of a Linux kernel 2.4.32 (the kernel that our static tool analyzed). The monitor loops over all the identified invariants and for each invariant variable, it compares the runtime value of the variable against its known-good value. The monitor emits a warning message if any comparison returns *false*. The list of invariants as well as their known-good values is derived by the static invariant analyzer presented in Section III.

One practical difficulty that our Invariant Monitor overcomes is the semantic gap between the monitor as a LKM and the rest of the kernel – not all global variables are exposed to the monitor as symbols. For example, `msg_ctlmax` is an unresolved symbol when we try to load the monitor. For this reason and for better portability (e.g., running the monitor from a hypervisor in the future), our monitor refers to the invariants by their runtime addresses rather than symbolic names. However, our static invariant analyzer reports invariants by names. Therefore, we need to bridge the gap between names and runtime addresses. To solve this problem, we use the information contained in the `System.map` file, a standard file generated during the kernel compilation process, which contains a mapping from kernel variable names to their runtime addresses.

A related difficulty brought by the semantic gap is the lack of offset information when our monitor makes fine-grain memory accesses to individual fields of structure variables or array elements, because `System.map` only provides the starting address of a structure or array variable but our monitor needs to look inside it. One naïve approach is to manually count the byte offset into a block of memory based on the type information, but this is not a scalable approach because our monitor needs to read hundreds of thousands of global variables (Table II). Instead, we leverage the power of static analysis to automatically generate code for the monitor. Specifically, the static analyzer generates code that declares pointer variables of the appropriate type, uses pointer dereferencing expressions to represent fine-grain memory accesses, and lets the compiler find out the correct offset information. For example, the static analyzer generates the code snippet in Fig. 4 for the invariant `timedia_data[3].num` where `timedia_data` is an array whose elements are of type `struct timedia_struct` that has a field named `num`. Here 0xc0272420 is the runtime address of the `timedia_data` array. Note that the known-good value 8 that is compared in the `if` statement is automatically supplied during the code generation because it is available to the static invariant analyzer by the time of code generation (i.e., in the *legal value list*).

2) Evaluation of false positives

To estimate the false positive rate of our invariant analyzer, we run the test programs in Table I while our Invariant Monitor is running in the background. The goal is to find whether the test programs can trigger legitimate updates to any global variable that our invariant analyzer believes to be invariant. If that's the case, our Invariant Monitor should generate warnings. While the non-existence of such warnings cannot be used as a proof that our invariants are all real, existence of such warnings does show that some of our invariants are false. In order to maximize the detection probability of false invariants, we choose the set of test programs in Table I that to our knowledge exercise all important subsystems of the kernel.

We ran these test programs after the kernel (version 2.4.32) was fully booted and our Invariant Monitor module was loaded. During the long time execution of the test programs, our checker reported warning messages about only one variable, which we categorize as a false positive by our invariant analyzer in the last row of Table III. This variable is `ipv4_devconf_dflt.rp_filter`. We carry out a manual investigation to understand why our invariant analyzer does not recognize legal assignments to this variable, and we find that our invariant analyzer misses it mainly due to a very subtle pointer arithmetic operation by the Linux kernel. We believe that this false positive can be eliminated by modifying our static analyzer. Overall, we are happy to see that our invariant analyzer has almost no false positives (1 out of 141,280 invariants monitored).

```
p=(struct timedia_struct*)0xc0272420;

if (((struct timedia_struct*)p)[3].num!=8)
 {printk(KERN_WARNING "Bad invariant
timedia_data[3].num \n");};
```

Figure 4. One example of automatically generated code for checking invariants at runtime

3) *Evaluation of false negatives*

Having false negatives means that our invariant detection does not recognize some variables that are actually invariants; if a rootkit manipulates such variables, our Invariant Monitor will not be able to detect it. One way to estimate the impact of false negatives is to run real-world rootkits on a system with our Invariant Monitor installed, and see whether our Invariant Monitor can detect them.

We selected several real-world rootkits for this purpose; they are Adore, Mood-nt, Phalanx-b6, Enyelkm-1.3, SuckIT 2, and Knark. Because the kernel version that we analyzed is relatively old (2.4.32), we were able to run only the SucKIT 2 rootkit. But our Invariant Monitor successfully detected this rootkit; the invariant that is violated is `sys_call_table[59]`, whose known-good value should be `sys_olduname`.

D. Performance of the invariant analyzer

All our experiments are performed on a server with a 2.93 GHz, 8-core Intel Xeon CPU and 16 GB of RAM. And our invariant monitor and runtime analyzer run in a virtual machine with 2GB of RAM and 20GB of disk, running a Linux 2.4.32 kernel.

Our static analyzer takes a merged Linux 2.4.32 kernel with 400,492 lines of C code as input, and produces the invariant report and the source code of the invariant monitor. The whole process takes 272 minutes and 4.3GB memory on average. In more detail, the pointer analyzer takes 151 minutes on average, and the invariant analyzer takes 121 minutes on average. Since the static invariant detection only needs to run offline, we have not aggressively optimized our static analyzer for speed.

E. Discussion

The degree to which a set of global invariants can approximate runtime integrity of a kernel remains a research question. For example, the invariants that we identified are all necessary conditions, but they may not be sufficient. Assuming that a right set of global invariants is at hand, we can estimate the runtime integrity of the kernel by verifying them. If all of them are verified, we have more confidence about the kernel's integrity. But if some of them do not pass the verification, we know that the kernel has lost its integrity.

V. RELATED WORK

Invariants detection

The Daikon invariant detector [13] generates likely invariants using program execution traces collected during sample runs. Daikon is a dynamic invariant detector, and its idea has been incorporated into Gibraltar [7] and ReDAS [18].

Integrity measurement mechanisms

There has been a long line of research on integrity measurement. Approaches such as IMA [25] use hashing or digital signatures to measure the software at load time. Recently, ReDAS [18] and DynIMA [12] advance the state of the art by supporting software integrity measurement at runtime. Other related work includes [14, 20, 22, 23, 24, and 32]. These approaches generally focus on the mechanism for measurement, but not the integrity properties.

Copilot [22] is a co-processor based integrity checker for the Linux kernel. The properties that Copilot prototype checked were kernel code, module code, and jump tables of kernel function pointers. Although Copilot later provided a specification language [23], its focus was not on deriving integrity properties. We work out the properties from analyzing the target software itself.

Livewire [14] leverages a VMM (a modified version of VMware workstation) to implement a host-based intrusion detection system. It can inspect and monitor the states of a guest OS for detecting intrusions, and interposes on certain events, such as interrupts and updates to device and memory state. Like Copilot, Livewire does not focus on the identification of integrity properties but only checks known properties.

LKIM [20] produces detailed records of the states of security relevant structures within the Linux kernel using the concept of contextual inspection. However, the identification of security relevant structures relies on domain knowledge.

Specialized integrity property measurement

Some specialized integrity properties have been measured, such as control flow integrity [4] and Information flow integrity [17]. CFI [4] checks if the control transfer from one function to the next is consistent with a pre-computed control flow graph, so we can think of it as checking a sequence property of the target software. PRIMA [17] checks the integrity of a system by reasoning about information flows. But it assumes that there is no direct memory modification attack, e.g., information flows are triggered by well-defined interfaces (function calls or file reads).

Rootkits detection and recovery

As we mentioned, there has been a lot of research on rootkits. A nice survey of rootkits and detection software is given in [22]. From [1] you can also find a list of popular rootkits. The integrity measurement mechanisms (such as [14, 22, 24, and 32]) mentioned above all can be used for rootkit detection. Some work such as [15] and [19] attempts to detect rootkits and recover the software from known-good copies.

Trusted computing

The Trusted Computing Group [28] has proposed several standards for measuring the integrity of a software system and storing the result in a TPM (Trusted Platform Module) [29] whose state cannot be corrupted by a potentially malicious host system. Industry vendors such as Intel have embedded TPM in their hardware. Such standards and technologies have provided the root of trust for secure booting [6], and enabled remote attestation [26]. There has been a consistent effort in building a small Trusted Computing Base (with hardware support such as TPM and application level technique such as AppCore [27]). A small Trusted Computing Base facilitates integrity analysis and monitoring.

VI. Conclusion

In this paper, we have studied the application of static source code analysis to derive integrity properties of an operating system kernel. We design and implement automated tools that can derive *global invariants* out of the target kernel without running it.

To evaluate our methodology, we apply our tools to the Linux kernel 2.4.32 and identify 141,279 global invariants that are critical to Linux's runtime integrity. Furthermore, we compare the invariant list generated by our static analyzer with the one generated by a dynamic invariant analyzer, and find a large number of variables that can cause false alarms for the dynamic analyzer. Our experience suggests that static analysis is a viable option for automated integrity property derivation, and it can potentially have very low false positive and false negative rates.

References

[1] Chkrootkit - rootkit detection tool. http://www.chkrootkit.org/.

[2] RootkitRevealer. http://technet.microsoft.com/en-us/sysinternals/bb897445.aspx.

[3] Sophos anti-rootkit. http://www.sophos.com/products/freetools/sophos-anti-rootkit.html.

[4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity", ACM Conference on Computer and Communications Security (CCS), Alexandria, VA, Nov. 2005.

[5] L. O. Anderson, "Program analysis and specialization for the C programming language", PhD thesis, University of Copenhagen, 1994.

[6] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture", In IEEE Computer Society Conference on Security and Privacy. IEEE, 1997, pp. 65–71.

[7] A. Baliga, V. Ganapathy, and L. Iftode, "Automatic inference and enforcement of kernel data structure invariants", In ACSAC '08: Proceedings of the 2008 Annual Computer Security Applications Conference, pages 77–86. IEEE Computer Society, 2008.

[8] A. Baliga, P. Kamat and L. Iftode, "Lurking in the shadows: identifying systemic threats to kernel data", IEEE Symposium on Security and Privacy, Oakland, CA, May 2007.

[9] P. Barham, B. Dragovic, K. Fraser, et al., "Xen and the art of virtualization", ACM Symposium on Operating Systems Principles (SOSP), Bolton Landing, NY, Oct. 2003.

[10] F. Bellard, "QEMU, a fast and portable dynamic translator", Proceedings of the 2005 USENIX Annual Technical Conference, 2005.

[11] Manuvir Das, Ben Liblit, Manuel Fähndrich, and Jakob Rehof, "Estimating the Impact of Scalable Pointer Analysis on Optimization", In Proceedings of the 8th International Symposium on Static Analysis (SAS'01), London, UK, 2001.

[12] L. Davi, A. Sadeghi, and M. Winandy, "Dynamic Integrity Measurement and Attestation: Towards Defense against Return-Oriented Programming Attacks", Proceedings of the 2009 ACM workshop on Scalable Trusted Computing (STC). November 2009.

[13] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants", In Science of Computer Programming, 2007.

[14] T. Garfinkel, M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection", Proceedings of Network and Distributed Systems Security Symposium (NDSS), February 2003.

[15] J. Grizzard, E. Dodson, G. Conti, J. Levine, and H. Owen, "Toward a trusted immutable kernel extension (TIKE) for self-healing systems: a virtual machine approach", Proceedings of 5th IEEE Information Assurance Workshop, 2004.

[16] Michael Hind. "Pointer analysis: haven't we solved this problem yet?" Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pp. 54-61.

[17] T. Jaeger, R. Sailer, and U. Shankar, "PRIMA: policy-reduced integrity measurement architecture", Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT 2006).

[18] C. Kil, E. Sezer, A. Azab, P. Ning, and X. Zhang, "Remote attestation to dynamic system properties: Towards providing complete system integrity evidence", Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'09), Lisbon, Portugal, 2009.

[19] J. Levine and J. Grizzard and H. Owen, "Re-establishing trust in compromised systems: recovering from rootkits that trojan the system call table", Proceedings of 9th European Symposium on Research in Computer Security, Sophia Antipolis, France, September 2004.

[20] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, C. D. McDonell, "Linux kernel integrity measurement using contextual inspection", Proceedings of the 2007 ACM workshop on Scalable Trusted Computing (STC). October 2007.

[21] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. "CIL: Intermediate language and tools for analysis and transformation of C programs", Conference on Compiler Construction (CC), Grenoble, France, Apr. 2002.

[22] N. Petroni, Jr., T. Fraser, J. Molina, W. A. Arbaugh, "Copilot—a coprocessor-based kernel runtime integrity monitor", 13th USENIX Security Symposium, San Diego, CA, Aug. 2004.

[23] N. Petroni, T. Fraser, A. Walters, and W. Arbaugh, "An architecture for specification-based detection of semantic integrity violations in kernel dynamic data", 15th USENIX Security Symposium, 2006.

[24] N. Petroni and M. Hicks, "Automated detection of persistent kernel control-flow attacks", 14th ACM Conference on Computer and Communications Security (CCS), Alexandria, VA, Oct. 2007.

[25] R. Sailer, X. Zhang, T. Jaeger, L. van Doorn, "Design and implementation of a TCG-based integrity measurement architecture", 13th USENIX Security Symposium, 2004.

[26] J. Sheehy, G. Coker, J. Guttman, et al. "Attestation: evidence and trust", http://www.mitre.org/work/tech_papers/tech_papers_07/07_0186/07_0186.pdf, accessed August 16, 2010.

[27] L. Singaravelu, C. Pu, H. Haertig, C. Helmuth, "Reducing TCB complexity for security-sensitive applications: three case studies", 1st ACM SIGOPS/EuroSys European Conference on Computer Systems, Leuven, Belgium, April 2006.

[28] Trusted Computing Group. http://www.trustedcomputinggroup.org, accessed August 16, 2010.

[29] Trusted Platform Modules. http://www.trustedcomputinggroup.org/developers/trusted_platform_module/specifications, accessed August 16, 2010.

[30] David Wagner. Static analysis and computer security: New techniques for software assurance. Ph.D. dissertation, Dec. 2000, University of California at Berkeley.

[31] Yi-Min Wang, Roussi Roussev, Chad Verbowski, Aaron Johnson, Ming-Wei Wu, Yennun Huang, and Sy-Yen Kuo. "Gatekeeper: Monitoring auto-start extensibility points (aseps) for spyware management". In LISA '04: Proceedings of the 18th USENIX conference on System administration, 2004.

[32] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer, "Secure coprocessor-based intrusion detection", Tenth ACM SIGOPS European Workshop, Saint-Emilion, France, September 2002.

[33] Iperf project page. http://sourceforge.net/projects/iperf/

[34] http://users.cis.fiu.edu/~weijp/Jinpeng_Homepage_files/report.xml (It is a huge file. Please open it with a text editor instead of the browser).