Eliminating Trust From Application Programs By Way Of Software Architecture

Michael Franz
University of California, Irvine

Abstract:

In many of today's application programs, security functionality is inseparably intertwined with the actual mission-purpose logic. As a result, the trusted code base is unnecessarily large and audit costs are high. We present a software architecture in which applications can be completely untrusted, even when they manipulate secrets. Key to our approach is the use of a trusted multi-level security virtual machine, inside of which all secrets remain locked at all times. In an experimental prototype, we were able to bring down the run-time overhead much lower than expected, by using aggressive dynamic compilation and static analysis techniques.

1 Introduction

Before the widespread adoption of databases, concepts such as "transactions," "commit," and "roll-back" had to be coded explicitly into application programs, and as a result often wound up interwoven with the actual business logic. Today, almost every major application program sits atop some kind of database system. Moreover, database interfaces have been largely standardized, making it possible to even swap out the database layer without a major rewrite of application code.

Interestingly, when it comes to information security, the state of the art today appears to be pretty much where it was with respect to data integrity 25 years ago: We find security functionality scattered all over application programs, and intermingled with the true mission-purpose functionality. For programs handling sensitive data, this implies an unnecessarily large trusted code base, simply because the critical parts cannot be separated from the non-critical ones. As applications grow larger and larger, the scope for errors is multiplied and auditing costs rise ever higher.

This leads to the question whether it would be feasible to do for information security what databases accomplished for information integrity. Can we pull all of the relevant functions out of application programs and move them into a standardized, exchangeable software layer below?

Of course, even today there are many well-structured applications that have been designed to minimize the proportion of the code that needs to be trusted. Our question, however, focuses on software design in general: **can we move trust out of application programs altogether**, so that even not-so-well designed applications cannot accidently (or maliciously) leak sensitive information? We answer this question in the affirmative and point to an architecture based on virtual machines.

A high-level virtual machine (VM) such as the Java VM is a good basis for an architecture in which untrusted programs can manipulate secrets, because the VM has absolute control over execution—an application program is merely *interpreted* on the VM, rather than being executed directly on some hardware, and all data accesses go through the VM. However, current VMs such as the Java VM lack a mechanism to differentiate between "secrets" and "non-secrets," and an efficient mechanism to prevent leaking of secrets through control flow.

We have constructed a prototype of a derivative of the Java VM that adds the missing functionality for true multi-level security programming. First, our VM adds a *security label* to every data item and prevents assignments that would leak secrets. Second, we *prevent security leaks via control flow* by attaching a security label to the program counter and coercing all data labels of assignment targets to the least upper bound of the assigned value and the current program-counter label.

Historically, such multi-level security (MLS) schemes [DoD85] have suffered from performance degradation and from the problem of "label creep," and as a consequence never reached widespread acceptance. Label creep refers to the effect that labels get coerced "upward" easily, while "downward" declassification is rare and difficult to do correctly. We were able to overcome these problems by employing compiler techniques that were either not available in the early days of MLS, or not feasible back then, particularly due to insufficient memory sizes.

Only recently have computer memories become large enough to support flow-sensitive whole-program analyses. Indeed, storage addressability is often still the limiting factor for this type of analysis today. However, due to the effects of Moore's law, most desktop machines will soon have sufficient RAM to routinely perform these kinds of analyses, and thereby make our scheme practical.

The remainder of this paper is structured as follows. First, we discuss current security architectures involving virtual machines. Then, we introduce a new *Multi-Level Security Virtual Machine* architecture. We summarize preliminary experience with a prototype implementation. We then discuss related work, followed by a short discussion and conclusions section.

2 Security Architectures Involving Virtual Machines

Building on significant research accomplishments that were made in the 1960's and 1970's [Gol69, GBE73, Gol74, PG74, PK75, KZB^{+}91], virtual machines are currently enjoying a major renaissance [FDF05]. One principal driver for this renewed interest in virtual machines is security, another is software portability.

There are two major categories of VM that are very different from each other. The first kind of VM (also often called a "low-level VM," "virtual machine monitor," or "virtualization platform") virtualizes a physical computer. Example contemporary VMs in this category are VMware [Wal02, SVL01] and Disco [BDGR97]. This kind of VM presents an interface exactly like, or very similar to, raw hardware, and one usually runs an ordinary operating system on top of it. This technology is now in the commercial mainstream. Very

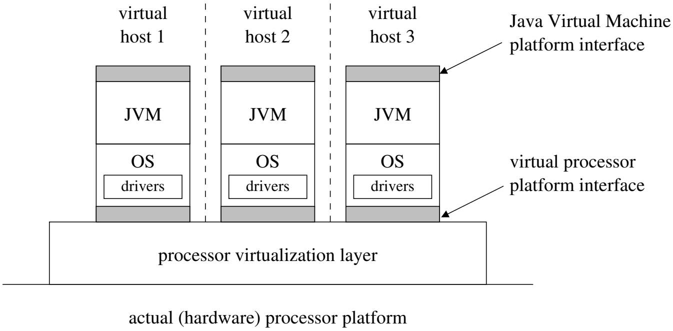
virtual host 1
virtual host 2
virtual host 3
Java Virtual Machine platform interface
JVM
JVM
JVM
OS
OS
OS
drivers
drivers
drivers
virtual processor platform interface
processor virtualization layer
actual (hardware) processor platform

Figure 1: Web hosting scenario: VM provides isolation for replicated software stacks.

closely related to low-level VMs are "hypervisor VMs" such as Xen [BDF$^+$03]. These present a hardware-close interface that requires modifications to the operating system, but not to application programs.

The second major category of virtual machine is the "high-level VM." Today's most popular representatives of this category are the Java VM and Microsoft's .Net Common Language Runtime (CLR). These virtual machines present an interface that is much richer than typical hardware interfaces, particularly in respect to the data types that are available. Because they abstract away from the actual hardware and operating system that is running on a computer system, high-level VMs have established themselves as the target platforms of choice for the Internet Computing paradigm. High level VMs usually enforce type safety, control safety, and memory safety, and thereby protect the host computer from malicious or faulty actions of the programs running on the VM.

Curiously, there are many computers today that are simultaneously running both kinds of VM. For example, in many web hosting scenarios, a customer purchases a "virtual host," a fraction of the physical resources of an actual host accessed through a "low-level VM." Customers are given the illusion of having their own dedicated remote computer, but in fact these are multiplexed onto fewer physical computers (Figure 1). The virtualization platform enforces complete separation between the various virtual hosts—in fact, the separation is so absolute that the complete software stack needs to be replicated on each of the virtual hosts, including the operating system, device drivers, and the high-level virtual machine that executes the users' web applications.

Another use of virtualization platforms has been to implement multi-level security (MLS) schemes [Rho75, Wei75, KLM88]. In this scenario, the different virtual hosts that are multiplexed onto a single physical computer have different security labels (Figure 2). There may also be specific hardware components that are available to only certain subsets of the virtual hosts, and not the others. For example, the different virtual hosts might be connected to different, disjoint computer networks and must be prohibited from talking to any but the appropriate networking hardware. As in the web hosting scenario, each of the virtual hosts will be running its own private copy of the device drivers and operating system, and if using platform-independent code, each will need its own private copy of a

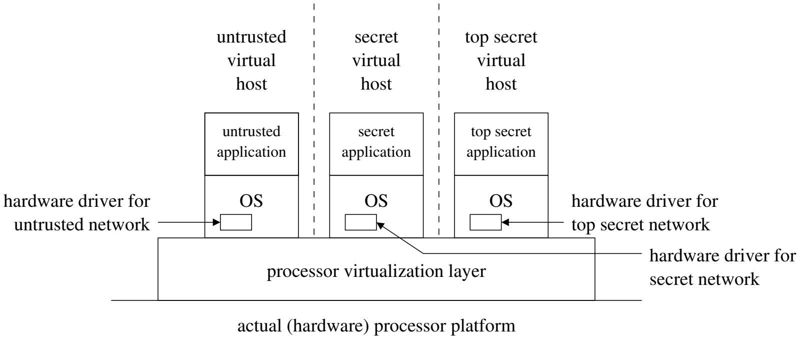
untrusted virtual host
secret virtual host
top secret virtual host
untrusted application
secret application
top secret application
OS
OS
OS
hardware driver for untrusted network
hardware driver for top secret network
hardware driver for secret network
processor virtualization layer
actual (hardware) processor platform

Figure 2: MLS Scenario: VM separates virtual computers with diverging security labels.

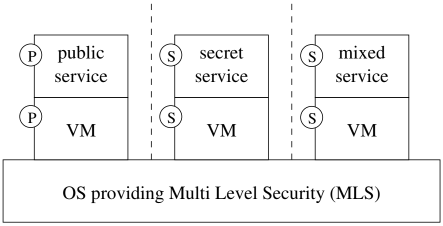

Figure 3: Using an OS that supports explicit security labels: applications still "mingle" secrets and non-secrets and must be labeled with the least upper bound of all the labels that can occur in a computation. P and S represent labels, with $P < S$.

high-level virtual machine such as the JVM. This security architecture is now deployed in certain government agencies.

Quite obviously, replicating the complete operating system stack as well as a high-level VM in this manner across multiple virtual hosts is not an elegant solution. However, it has the advantage that an off-the-shelf OS and an off-the-shelf high-level VM can be used, and it keeps a small "low-level VM" in charge of the separation. One might then argue that this virtual machine monitor forms the trusted code base as far as separation between the labels is concerned, but in each virtualization compartment taken separately, the OS and high-level VM are still part of a "trusted code base."

A somewhat better solution would be to use an operating system that explicitly supports security labels, such as SELinux [SVS02] or Solaris Trusted Extensions [Sun]. In this scenario (Figure 3), which represents the current state of the art, the "multiplexing" between different security levels needs to occur only above the OS layer, because the OS can separate between "secrets" and "non-secrets" internally. However, every application program still needs to carry a label that represents the least upper bound of all the labels it manipulates. If the architecture contains a high-level VM (for example, the common scenario of running Java services on top of Java VMs), then that high-level VM must also be replicated for every security label.

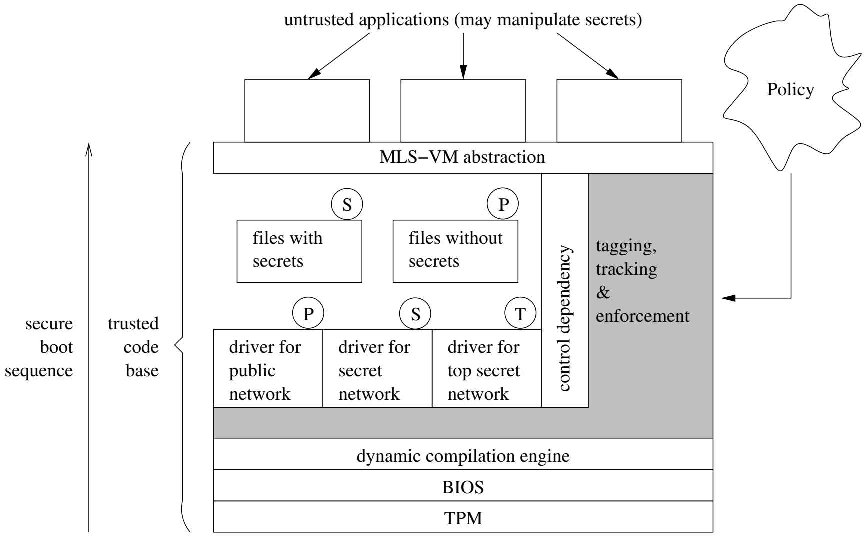
untrusted applications (may manipulate secrets)
Policy
MLS−VM abstraction
S
P
files with secrets
files without secrets
P
S
T
driver for public network
driver for secret network
driver for top secret network
control dependency
tagging, tracking & enforcement
dynamic compilation engine
BIOS
TPM
secure boot sequence
trusted code base

Figure 4: MLS-VM locks all data inside and mediates all information flows, including those that occur implicitly via control flow. The MLS-VM constitutes the trusted code base; all application programs are completely untrusted, even those that manipulate secrets.

Note that when we say that an application "is labeled with the least upper bound of the labels it manipulates," then that really means that the application **is trusted** up to that label. There is no general way we can prevent such an application (if it is malicious or faulty) from leaking information between different channels it has simultaneous access to. Hence, while the use of a multi-level security operating system can surely limit damage (for example, by making sure that no "secret" data is ever read by a program that has only "non-secret" access), this architecture still requires **trust** in programs that have simultaneous access to multiple channels. As a consequence, these programs need to be audited to the standard implied by the highest label they manipulate.

3 MLS-VM: A New Security Architecture

A new security architecture removes this obligation. In this architecture, **completely untrusted application programs** can be allowed to perform computations on **sensitive data** without the risk that any of the secrets would ever be leaked. Our approach takes the core ideas of multi-level security schemes as they have been successfully used in operating systems for decades, and brings them to the semantically much richer realm of high-level virtual machines. The result is what we call a Multi-Level Security Virtual Machine, or MLS-VM.

In our model, the MLS-VM forms the uppermost layer of the trusted code base on the target platform. In a production environment, its integrity would need to be established using techniques that have already been developed, such as secure boot [AFS97] using a

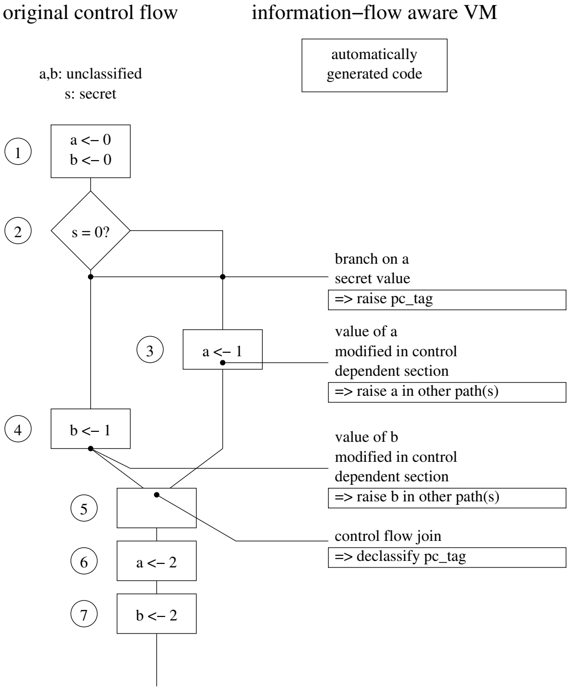
original control flow
information–flow aware VM
automatically generated code
a,b: unclassified
s: secret
1
a <– 0
b <– 0
2
s = 0?
branch on a secret value
=> raise pc_tag
3
a <– 1
value of a modified in control dependent section
=> raise a in other path(s)
4
b <– 1
value of b modified in control dependent section
=> raise b in other path(s)
5
control flow join
=> declassify pc_tag
6
a <– 2
7
b <– 2

Figure 5: Hybrid information-flow enforcement combines load-time static analysis and code rewriting with dynamic run-time techniques.

Trusted Platform Module (TPM) [DLP$^+$01, JSM01, Smi02, ML05]. Once that the MLS-VM is running, it can give untrusted programs access to sensitive data while enforcing information-flow policies. The MLS-VM achieves this goal by internally tagging every single data item, tracking the tags through the lifetime of each data item, and by enforcing policies on the interactions of data items among themselves and with the boundaries of the VM. Such boundaries include the file system and the network.

Hence, the MLS-VM (Figure 4) achieves far more than simply lifting the separation architectures of Figures 1, 2, and 3 to a higher level in the software stack. The MLS-VM actually allows to write a single, **untrusted** application program that simultaneously has access to channels with different security labels, and still can guarantee strict **non-interference** [GM82]. Non-interference is the property that “secret” inputs to a program never interfere/change the “publicly observable” values in the program. We explicitly exclude timing channels [McH95] from our consideration since they reside “out of band” relative to our approach; these will have to be dealt with using appropriate “out of band” countermeasures such as processor clock-frequency variation.

Note that enforcing correct information flows is considerably more complicated than merely prohibiting direct assignments of “high-label” values into “low-label” variables; one also needs to consider *implicit* flows that are carried by a program’s flow of control. Consider the program depicted on the left in Figure 5: Here, a branch is taken on a secret value. Any store to a "low" value in a region that is dependent on such a branch may leak information, unless the store is dead (i.e., later overwritten without reading the previous value). Even more subtly, non-execution of a certain branch may leak as much information as taking the branch. Traditionally, dynamic information-flow techniques have had trouble dealing with this kind of situation because they only follow a single control-flow path rather than considering all alternative paths simultaneously.

Our approach uses a combination of static analysis and dynamic tagging, tracking, and enforcement techniques (in concert with aggressive just-in-time compilation) to handle control-dependent information flows. The key to our solution is to perform static analyses with binary rewriting ahead of actual execution on the MLS-VM. These analyses automatically insert compensating tag instructions into alternative paths whenever any variable is modified along just one path.

In the example of Figure 5, branching on a `secret` value raises the label of the program counter to the least upper bound of its current value and `secret`. Every variable that is modified has its own label coerced to the least upper bound of its current label and the label of the program counter. For example, the assignment to the non-secret variable a in basic block 3 will raise the label of a to `secret` because the program counter is `secret`, preventing a leak of the contents of variable s via a. In spite of the coercion of a's label, we would still be able to infer the value of s in basic block 5 by observing the non-secret variable b. This is prevented in our implementation by automatically inserting compensation code, so that when any variable is modified in *any* branch, its label is coerced in *all* branches. The right side of Figure 5 shows the instructions that are automatically inserted after static analysis. In particular, b is coerced to `secret` in basic block 3 and a is coerced to `secret` in basic block 4.

When the control flow re-joins in basic block 5 (the immediate post-dominator of blocks 3 and 4), the program counter's label can be restored to the value it had before the branch in basic block 1. Note that a and b remain classified past this control-flow join; a will be declassified in basic block 6 and b in basic block 7, i.e., at the points that any data values that could reveal the value of s are dead.

4 Preliminary Experience With an MLS-VM

In the course of the past four years, we have implemented two increasingly complex prototypes of Java VMs that (in addition to the full functionality of a standard Java VM) provide comprehensive information-flow-enforcement infrastructures. Our first implementation [HCF05] realizes only part of the MLS-VM vision: it performs labeling at the granularity of objects rather than individual instance variables, and it does not track information leaks via control flow. Our second implementation [CF07] is considerably more ambitious and pushes the labeling mechanism down to the level of individual data fields. It also correctly handles control flow as explained in the previous section. As a result of these additional capabilities, the second implementation has a considerably higher runtime overhead.

In both of the implemented systems, the new capabilities added to the Java VM are entirely directed by policies that themselves reside outside of the VM and that are dynamically changeable at will. These late-bound policies govern the functions of the labeling and enforcement mechanism as follows:

- *Tagging:* A newly created object and its instance variables are assigned labels. What labels are assigned in particular depends on the context of the object's creation (including the label of the program counter) and an external policy.
- *Tracking & Enforcement:* Any interaction between two objects is governed by policies. Among such interactions are instance variable accesses and method invocations. Method calls result in indirect information flow through parameters and return values. A policy determines whether or not the interaction is allowed, and if yes, if any tags are modified as a result of the interaction. If the interaction is disallowed, an exception is raised.
- *Output Channels:* Of particular interest are object interactions at the system's boundary, e.g. with the file system or the network interface. Such interactions can now be controlled based on the label of the information that wants to be written to a file or to the network.

The surprising result is that the cost of performing these labeling operations at the granularity of *whole objects* (first implementation) throughout a full Java VM using the technique of binary rewriting of the bytecode at load-time amounts to only about 6% overhead after only very rudimentary tuning.

But even when performing labeling at the granularity of *individual fields* (second implementation), the overhead is still remarkably low. In our measurements using the Java Grande [MCH99] benchmarks, among the Section 1 microbenchmarks that measure primitive operations, only the assignment operation stands out with a slowdown factor of slightly more than 3 (Figure 6). This makes sense, because it is during assignment that labels need to be calculated and modified. However, few realistic programs consist entirely of assignments. Looking at Java Grande Section 2, which contains library kernels, the largest slowdown is just slightly more than a factor of 2.5. And looking at Java Grande Section 3, which contains larger applications representative of "real world" programs, the overhead of our scheme (Figure 8) represents a slowdown factor of about 2 in the worst case. In many security-relevant contexts, people would probably gladly accept such a slowdown in exchange for information-flow security, or simply wait three years until Moore's law had erased the costs.

It is important to note that our approach does not require any immediate changes to existing software that already runs on the underlying VM platform (in our case, the Java VM). While eventually one would probably stop writing application programs that intertwine the mission-critical and security functionalities, the immediate benefit of a security layer such as ours is that it can guarantee correct information flows even if the applications running on top are faulty or malicious. As long as application programs behave correctly, our system is completely invisible. However, the cost of auditing applications disappears because we can now automatically detect and abort misbehaving programs.

Second, our approach does not hard-code any particular policy. In fact, the labeling mechanism we have implemented and that we propose as a general architectural approach is completely policy-agnostic. While it can be used to implement standard Bell-LaPadula style [BL73] information-flow control, the exact same mechanism can be used for other information-flow schemes. For example, we have successfully implemented a Perl-style taint propagation for Java on top of our basic infrastructure [HCF05]. This allows us to fix faulty web applications *after the fact*. No modification of any application program is required—all correct programs will continue to function exactly as before. However, if there is some path in some application program along which a tainted input can reach a critical sink such as an SQL database query, and that path is triggered by an actual execution under our modified VM, then the VM will detect this fact and throw an exception. We tested our framework with a well-known suite of vulnerable web application samples and our framework was able to prevent the attacks that these sample applications are vulnerable to.

Third, our system provides completely dynamic policies that can be changed even while a program is already running. This is in contrast to previous approaches such as Myers' Jflow and Jif [Mye99a, Mye99b] that extend the Java language with statically checkable information-flow annotations. At compilation time, the information-flow policy then becomes "frozen." In Erlingsson and Schneider's approach [ES99, ES00], a security automaton [Sch00] is in-lined into each program prior to execution, which again "freezes" the security policy; furthermore, this solution is limited by what is decidable by a security automaton. In our system, the policy is entirely separate from the enforcement mechanism and remains separate and modifiable during program execution.

5 Related Work

Information security is important, and hence there is a large body of related work. Of particular relevance to our work is previous research in system dependability, information flow and static analysis. Due to space constraints, we can address only a small selection of related work that is most relevant to particular issues mentioned in this paper.

The software engineering community has studied a wide range of dependability issues [Kni04], of which information security (a combination of confidentiality and integrity) is just a subset. In fact, one could argue that we are only able to make confidentiality and integrity a main focus today because the community has been so successful at getting a grip on the remaining dependability properties (reliability, availability, safety, and maintainability), which has enabled the construction of large networked information systems in the first place.

In his survey of dependability assessment of software-base systems, Littlewood [Lit05] observes that the main focus in the past has been on reliability and safety, and that the field of *security* assessment is still in its infancy. A system such as ours would make it possible to exclude the application program from such an assessment and focus on the MLS-VM layer and the correct specification of policies.

Leveson et al. [LSST83] used the term "safety kernel" to describe an architecture in which the safety-relevant mechanisms are concentrated in a centralized location. We have used the more modern term "trusted computing base," but our approach is very much in the spirit of their pioneering work.

Bell and LaPadula [BL73] pioneered the use of a state machine to model security policies that specify security levels for data, and access rules for users with different clearance levels. Every event in a system is mapped to a transition in a corresponding state machine. Safety of a system is ensured by allowing transitions only to secure states. A secure state is defined as one in which the user has adequate clearance, as defined by the security policy, to access the data. The model also ensures data integrity by only allowing processes or users with the same clearance level to perform destructive writes to an object. Non-destructive writes are allowed to low-level clearance processes as long as this does not lead to an information leak.

Denning [Den76] extended the Bell-LaPadula model to use a lattice for sensitivity labels, with labels higher up in the lattice being more sensitive. Denning's model allows data to be relabeled only to a label higher up in the lattice, thereby ensuring that information always flows to a label that is at least as sensitive as the current one. Denning was also one of the first to point out that the information flow property should be enforced statically to contain label creep, and to avoid leaks through implicit flows.

More recently, the non-interference property has been studied and formulated in terms of type systems. Volpano et al. [VS97] formalize the soundness of Denning's analysis by developing a type system that is equivalent to the rules proposed by Denning. They then prove that this type system observes non-interference. Banerjee and Naumann [BN02] extend the scope of Volpano's work to encompass data-flow via mutable object fields and control-flow in dynamically dispatched method calls. The non-interference property is proved in much richer context with constructs of pointers and mutable state, private fields and class-based visibility, dynamic binding and inheritance, casts and type tests, and mutually recursive classes and methods.

Bernardeschi and et al. [BFL02] use type-based abstract interpretation (which is similar to bytecode verification) to prove information flow safety of Java bytecode. They, like Denning, handle implicit flows and make use of the immediate post-dominator relation to declassify the security label of the execution context. Our approach is different from their purely static analysis as we use both dynamic and static techniques, which makes the analysis more flexible and precise.

Several research projects apply static analysis to C programs. Evans' Split static analyzer [EL02] takes as input C source code annotated with "tainted" and "untainted" annotations. This is accompanied by rules for how objects can be converted from one into the other, and which functions expect what kinds of arguments. Shankar et al [STFW01] use a similar approach in which C source code is annotated, but they use type qualifiers instead.

The WebSSARI [HYH$^+$04] project analyzes information flow in PHP applications statically. It inserts runtime guards in potentially insecure regions of code. It differs from approaches such as Myers' Jflow and Jif [Mye99a, Mye99b] in that it does not require source annotations.

RIFLE [VBC$^+$04] is a system that tracks information flow dynamically. This is accomplished by using a combination of hardware and software. The underlying hardware architecture is modified to have labels for each register and storage word. At load time, binaries are rewritten to make implicit flows explicit using a reaching analysis. This work comes closest to our approach; however, the major difference between RIFLE and our system is the semantic level at which labeling occurs. Because RIFLE analyzes native binaries, it needs to be very conservative in its reachability analysis. Our approach uses a high-level virtual machine with strong memory safety guarantees and builds information-flow on top of semantically much richer abstractions.

6 Discussion And Conclusion

Most contemporary approaches to multi-level security are focusing on the operating system layer. For example, SELinux [SVS02] and Solaris Trusted Extensions [Sun] associate labels with OS-level abstractions such as files and sockets, and assign labels to whole applications. An application program that needs to manipulate data with multiple labels will itself require a label that is the least upper bound of all the labels occurring in the computation.

Conversely, the use of virtual machines makes it possible to distinguish labels at a much finer granularity. A VM environment has two unique advantages over executing binary code: (1) applications execute under full control of a VM that is itself part of the trusted code base, and (2) rather than manipulating data directly, an application running atop a VM really only manipulates *capabilities* to data, while the data itself remains “locked” inside the VM at all times. The combination of these properties leads to the idea of an MLS-VM that retains the advantages of “high-level” virtual machines such as the Java VM and Microsoft’s .Net Common Language Runtime, and combines them with information labeling, information-flow tracking, and policy enforcement capabilities. This in turn makes it possible to move all application programs (running atop a VM now) out of the trusted code base.

Unlike typical binary instruction set architectures that provide direct support for only a very small set of data types (and in particular cannot distinguish between integers and pointers), virtual machines can enforce strong type safety, memory safety, and referential integrity. This in turn can be used as a foundation for layering true information-flow controls on top.

Of course, a virtual machine is in itself a substantial piece of software, perhaps raising concerns about having it serve as a trusted computing base. In the scenario we have described, however, the effort of certifying the VM layer could be amortized across a very large number of users and application programs, making it cost-effective to apply auditing standards that are commensurate with high-assurance software.

Architectures such as the we have presented here will simplify the process of developing software that simultaneously handles sensitive and non-sensitive information. Currently, such software must be part of the trusted code base. Unfortunately, our trust in this type of software is often misplaced. For example, many “phishing” attacks have exploited errors in web browsers that allowed sensitive information inside the web browser to be sent to unintended remote parties [Fra07]. Our MLS-VM is able to prevent such information flows even if a web browser were *malicious rather than merely faulty*. In our approach, **application programs need never be trusted**.

7 Acknowledgement

Parts of this effort have been sponsored by the National Science Foundation under grants TC-0627747 and TC-0209163, and by the United States Homeland Security Advanced Research Projects Agency (HSARPA) and Air Force Research Laboratory (AFRL) under agreement number FA8750-05-2-0216. The author is also extremely grateful for generous unrestricted gifts supporting his research from Intel, Microsoft Research, Mozilla Corporation, and Sun Microsystems.

Any opinions, findings, and conclusions or recommendations expressed here are those of the author and should not be interpreted as necessarily representing the official views, policies or endorsements, either expressed or implied, of HSARPA, AFRL, NSF, any other agency of the United States Government, or any of the companies mentioned above.

References

[AFS97] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *1997 IEEE Symposium on Security and Privacy*, pages 65–71, 1997.

[BDF$^+$03] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[BDGR97] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.

[BFL02] C. Bernardeschi, N. De Francesco, and G. Lettieri. Using standard verifier to check secure information flow in Java bytecode. In *26th International Computer Software and Applications Conference*, pages 850–855, 2002.

[BL73] D. E. Bell and L. J. LaPadula. *Secure Computer Systems: Mathematical Foundations*. Technical Report Report MTR 2547 v2, MITRE, 1973.

[BN02] A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *15th IEEE Computer Security Foundations Workshop*, page 253, 2002.

[CF07] D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a Java virtual machine. In *23rd Annual Computer Security Applications Conference*, 2007.

[Den76] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[DLP$^+$01] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart. Building the IBM 4758 Secure Coprocessor. *IEEE Computer*, 34(10):57–66, 2001.

[DoD85] DoD United States Department of Defense. *Trusted Computer System Evaluation Criteria, DoD Standard 5200.28-STD*. 1985.

[EL02] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, Jan/Feb, 2002.

[ES99] U. Erlingsson and F. B. Schneider. SASI enforcement of security policies: a retrospective. In *New Security Paradigms Workshop*, pages 87–95, 1999.

[ES00] U. Erlingsson and F. B. Schneider. IRM Enforcement of Java Stack Inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, 2000.

[FDF05] R. Figueiredo, P. A. Dinda, and J. Fortes. Guest editors' introduction: Resource virtualization renaissance. *IEEE Computer*, 38(5):28–31, 2005.

[Fra07] M. Franz. Containing the ultimate Trojan Horse. *IEEE Security and Privacy*, 5(4):64–68, July 2007.

[GBE73] U. O. Gagliardi, L. Bolliet, and R. P. Goldberg (Eds.). *Workshop on Virtual Computer Systems*, 1973.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[Gol69] R. P. Goldberg. *Virtual Machine Systems*. Technical Report MS-2686, MIT Lincoln Laboratory, 1969.

[Gol74] R. P. Goldberg. A survey of virtual machine research. *I.E.E.E. Computer*, 7(6):34–45, 1974.

[HCF05] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for Java. In *21st Annual Computer Security Applications Conference*, pages 303–311, 2005.

[HYH$^+$04] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *13th International World Wide Web Conference*, pages 40–52, 2004.

[JSM01] S. Jiang, S. Smith, and K. Minami. Securing web servers against insider attack. In *17th Annual Computer Security Applications Conference*, pages 265–276, 2001.

[KLM88] P. A. Karger, T. E. Leonard, and A. H. Mason. *Computer with Virtual Machine Mode and Multiple Protection Rings*, U.S. Patent No. 4787031, November 1988.

[Kni04] J. C. Knight. An introduction to computing system dependability. In *26th International Conference on Software Engineering*, pages 730–731, 2004.

[KZB$^+$91] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering*, 17(11):1147–1165, 1991.

[Lit05] B. Littlewood. Dependability assessment of software-based systems: State of the art. In *27th International Conference on Software Engineering*, pages 6–7, 2005.

[LSST83] N. G. Leveson, T. Shimeall, J. Stolzy, and J. Thomas. *Design for Safe Software*, 1983.

[McH95] J. McHugh. Covert Channel Analysis, Technical Memorundum 5540:080A, Naval Research Laboratory, Washington D.C., 1995. A Chapter of the *Handbook for the Computer Security Certification of Trusted Systems*, 1995.

[MCH99] J. A. Mathew, P. D. Coddington, and K. A. Hawick. Analysis and development of Java Grande benchmarks. In *ACM Java Grande Conference*, pages 72–80, 1999.

[ML05] J. P. McGregor and R. B. Lee. Protecting cryptographic keys and computations via virtual secure coprocessing. *SIGARCH Computer Architecture News*, 33(1):16–26, 2005.

[Mye99a] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[Mye99b] A. C. Myers. *Mostly-static Decentralized Information Flow Control.* PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1999.

[PG74] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[PK75] G. J. Popek and C. S. Kline. The PDP-11 virtual machine architecture: A case study. In *Fifth ACM Symposium on Operating Systems Principles*, pages 97–105, 1975.

[Rho75] R. Rhode. *Secure Multilevel Virtual Computer Systems.* Technical Report ESD-TR-74-370, MITRE Corp., Bedford, Massachussetts, 1975.

[Sch00] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[Smi02] S. W. Smith. Outbound authentication for programmable secure coprocessors. In *7th European Symposium on Research in Computer Security*, pages 72–89, 2002.

[STFW01] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium*, pages 201–220, 2001.

[Sun] Sun Microsystems, Inc. *Trusted Solaris Operating System.*

[SVL01] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference*, pages 1–14, 2001.

[SVS02] S. Smalley, C. Vance, and W. Salamon. *Implementing SELinux as a Linux Security Module*, NAI Labs Technical Report. Technical report, May 2002.

[VBC$^+$04] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An architectural framework for user-centric information-flow security. In *37th International Symposium on Microarchitecture*, pages 243–254, December 2004.

[VS97] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *10th Computer Security Foundations Workshop*, pages 156–168, 1997.

[Wal02] C. Waldspurger. Memory resource management in VMware ESX server. In *Fifth Symposium on Operating Systems Design and Implementation*, pages 181–194, December 2002.

[Wei75] C. Weissman. Secure computer operation with virtual machine partitioning. In *National Computer Conference, Anaheim, California*, pages 929–934, 1975.

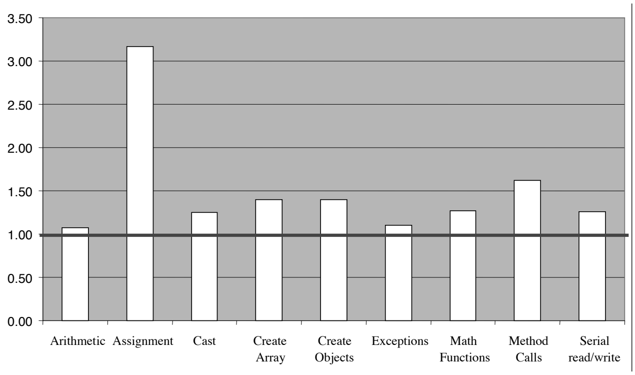

Figure 6: Java Grande benchmarks, Section 1 (primitive operations); normalized slowdowns relative to the unmodified benchmarks.

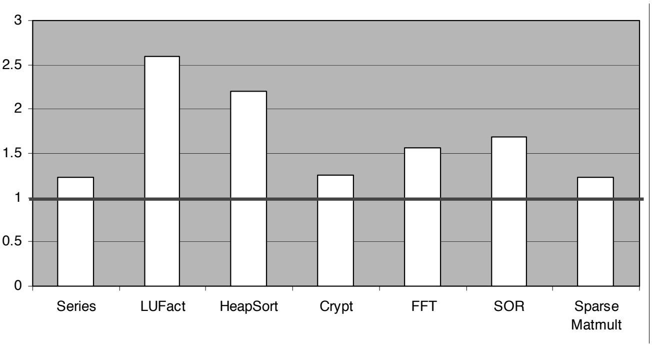

Figure 7: Java Grande benchmarks, Section 2 (library kernels); normalized slowdowns relative to the unmodified benchmarks.

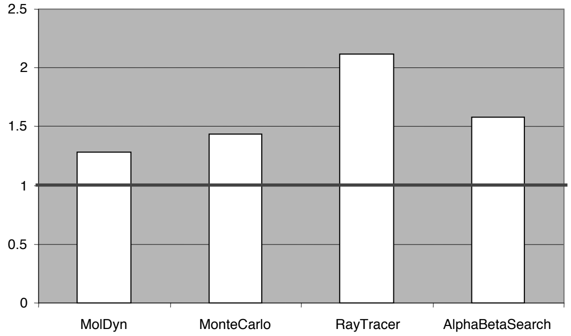

Figure 8: Java Grande benchmarks, Section 3 (large applications); normalized slowdowns relative to the unmodified benchmarks.