Engineering Sufficiently Secure Computing

Brian Witten
Symantec Research Labs
bwitten@symantec.com

Abstract

We propose an architecture of four complimentary technologies increasingly relevant to a growing number of home users and organizations: cryptography, separation kernels, formal verification, and rapidly improving techniques relevant to software defect density estimation. Cryptographic separation protects information in transmission and storage. Formally proven properties of separation kernel based secure virtualization can bound risk for information in processing. Then, within each strongly separated domain, risk can be measured as a function of people and technology within that domain. Where hardware, software, and their interactions are proven to behave as and only as desired under all circumstances, such hardware and software can be considered to not substantially increase risk. Where the size or complexity of software is beyond such formal proofs, we discuss estimating risk related to software defect densities, and emerging work related to binary analysis with potential for improving software defect density estimation.

1. Introduction, purpose, and motivation

Perfection is not often tractable in real systems, and there are no silver bullets in the increasingly high-stakes move-counter-move game titled "Computer Security." However, it is still possible to continue making systems increasingly stronger, and it should be possible to make systems strong enough to effectively neutralize broad categories of malicious actors seeking to break or subvert computer systems. It might even be possible to create technology effectively neutralizing so many malicious actors that the number remaining are so few that national and international efforts above and beyond technology might be effective in neutralizing them. However, even if such progress might be tractable, such progress is a long distance from the exponentially growing billion dollar per year problem of online identity theft, a long distance from fifteen terabytes of information taken from a single military network, and a long distance from today's multi-billon dollar problem of intellectual property leakage.

1.2. Outline of paper

Cryptography stands among the oldest and strongest techniques for protecting information. However, effective deployment can be challenging and cryptography is not without limits. Strengths challenges and limitations of cryptography are reviewed quickly in Section 2. Because information often must be decrypted and exposed for processing to be processed effectively, most techniques for protecting information in processing involve separating the exposed information from other information and other factors so that the information is exposed to only a limited set of threats. These techniques are discussed at length in Section 3. Separation Properties where the paper describes advantages of separation kernels. These advantages apply to permanent domains of intranets, extranets, safety critical systems, the open Internet, personal domains for finance, friends and family, and the many ad hoc domains necessary for managing national scale emergencies, and international collaboration managing natural disasters and transnational threats. Additionally, because these separation properties are critical to protecting information in scalable systems, it may be useful to verify the separation properties through formal verification techniques. These techniques, their limits, and the arguments for formal verification are discussed at length in Section 4. Formal Verification. Unfortunately, the most thorough formal verification techniques do not currently scale to formal verification of all properties of all systems. In fact, given scale and complexity of most software systems today, with the full functionality and complexity often necessary to meet "mission,"

"operational," "business," or "consumer" needs, the most thorough forms of formal verification do not even scale to the size of most software systems today. For this reason, we consider other approaches for gauging the degree to which such large and complex software satisfies security requirements. The other approaches considered include lighter forms of formal verification and other evaluation techniques, such as testing. Specifically, in Section 5, the paper proposes to leverage emerging research in model checking and binary analysis to improve software defect density estimation, and describes some of the limits and dangers in using such a gauge. Section 6 proposes a framework for reasoning about transitive risks between security domains that are strongly separated through cryptographic separation and separation kernels except for carefully specified interfaces, and Section 7 provides a summary.

2. Cryptography: advantages and hurdles

Where utilized effectively, strong cryptography has long provided sufficient security for transmission and storage of information. However, it is not yet pervasive in protection of valuable information for several reasons. Infrastructures for managing a large number of cryptographic keys are generally cumbersome and expensive. Encrypted information is hard to search, and any fragility of the key management infrastructure can have a strong impact on the recoverability of information. Cryptography burdens the system with performance impact. Also, encrypted communications and encrypted storage provide negligible value when the information is readily taken from relatively weak operating systems whenever exposed for processing. However, with the emergence of high assurance separation kernels [1, 7] as described in Section 2.0, and continued growth of excess performance capabilities of hardware, many of these concerns are diminishing.

However, cryptography has profound advantages. Cryptography not only protects secrecy of information in storage and transmission, but cryptography may also be used to protect the integrity and availability of information in storage and transmission. Clearly cryptography helps protect the integrity of information when cryptographic keys are needed to modify information without the tampering being immediately obvious. Also, because cryptography allows relatively secure use of untrustworthy storage, cryptography can be used to relatively safely replicate information into a larger number and wider variety of less trustworthy storage sites. In this manner, cryptography can increase the probability of confidential information retrieval in the wake of natural faults, and also increase the number of malicious actors required to collude to deny information retrieval. Similarly, cryptography can be used to increase availability of communications by increasing the number and diversity of communication paths available. In these ways, cryptography can be used to protect secrecy, integrity, and availability of information in transmission and storage. However, protecting information exposed for processing is a different matter entirely.

3. Separation Properties

As mentioned above, protecting information exposed for processing is quite different from protecting information in storage or transmission, and most techniques for protecting information in processing involve separating the exposed information from other information and other factors so that the information is exposed to only a limited set of threats. Traditionally, several techniques have been used to maintain the separation desired to protect information in processing. These include:

a) Labor-intensively attempting to establish and maintain networks that have no connectivity with other networks. Information within the network is exposed to threats from all other network participants, but theoretically the threat should be bounded to the participants of the "isolated" network. However, in practice most commonly, strong mission, operational, business, or consumer needs exist to connect these networks to other networks, invalidating the assumptions of bounded risk.
b) This connectivity between "otherwise isolated" networks is often established through a restricted set of protected "guards" that serve as a gateway from one network to another. This focuses the problem of controlling separation into a limited number of machines, and it is very common to use techniques of (c) and (d) to maintain such separation. However, anytime bits flow across boundaries, the overt and covert channels introduce transitive risks that are hard to manage because they directly violate the desired separation. Even if the overt and covert channels are small, the risk can be very large, particularly if software is being added or updated in the protected network in large or frequent increments from potentially compromised software development lifecycles. Furthermore, like a large balloon, where relatively high volume guards protect relatively

large networks, a single pin-prick mistake in the sealing of the large network can allow very large volumes to be exfiltrated from the breadth of the network over time. This argues for smaller security domains with stronger internal protections so that a flaw in one corner of the network does not devastate all participants.

c) Several techniques attempt to monitor information as it is being processed in a shared computing environment to ensure that bits of information do not influence each other, even if the bits of information exist in a shared environment such as a shared processor, shared memory, or other shared resource. These techniques include tainting, sandboxing, and monitoring the operating system (OS) to ensure applications aren't tampering with the OS. More broadly, these techniques include many types of current host based intrusion detection, intrusion prevention, antivirus, anti-spyware, and kernel integrity monitoring. However, many of the arguments for the integrity and trustworthiness of these techniques are based on unproven and potentially invalid assumptions about the integrity of the underlying OS, hardware, and protections afforded by the hardware and OS. Some of the techniques which do not make such assumptions are described in Sections 3.3 and 3.6.

d) Last, where machines must participate in multiple security domains, either as guards or as workstations facilitating end-user participation in multiple security domains, use of virtualization is an active area of research to facilitate a physical machine hosting multiple virtual machines with each virtual machine isolated from the other virtual machines and participating in a different security domain. [2, 3, 4, 5] Given the limits and dangers of the other approaches described above, this approach and the path for improving it are foci of this paper. This approach has the potential to offer security greater than the approaches described in (c), security improving to rival (a), particularly given the risks from "workarounds" and "sneakernet." Moreover, the flexibility of this approach can dramatically exceed the flexibility of (a), exceed the flexibility of (b), and rival the flexibility of (c).

3.1. Hypervisors and separation kernels

Virtualization based on separation kernels [6, 7] has tremendous advantages over the hypervisor [8 , 9] approach currently being pursued with Linux and Microsoft operating systems.[2, 8] Before describing the differences between hypervisors and separation kernels and advantages of separation kernels over hypervisors, it is important to note first that separation kernels do not provide the virtualization for full functional operating systems to run directly atop the separation kernel. However, virtualization can be built cleanly and severably atop the separation kernel and below the guest OS, and virtualization constructed in that manner can provide full virtualization for full functional operating systems to run with their full functionality.[5] Unlike para-virtualization, this can be done without requiring modification of the guest OS.[10] This has been demonstrated by Green Hills Software.[5] Such clean segregation of virtualization code from the code enforcing separation properties enables a dramatically smaller Trusted Computing Base (TCB), often with a security critical codebase that is smaller than 10,000 lines of code, and solidly evaluatable through formal verification. In contrast, monolithic hypervisors, even "micro" hypervisors such as VMware entangle the virtualization and separation properties into codebases that most commonly near a hundred thousand lines of code, and lack clean segregation of any TCB within that codebase. In contrast, the separation kernel is roughly between 200kb and 500kb. It is important to note that the security properties of some but not all separation kernels have been formally verified, and depending upon a separation kernel whose security properties have not been verified may have hidden risks. The security strength of formally verified separation kernels is the primary advantage of separation kernel based virtualization over hypervisors. However, separation kernel based virtualization additionally has several other advantages over hypervisors.

In addition to introducing less security risk, separation kernel based virtualization can also provide greater functionality. For instance, because the virtualization for each virtual machine is separated by the separation kernel from the virtualization underlying the other virtual machines, separation kernels naturally support parallel virtualization of different hardware simultaneously. In other words, on a single arbitrary x86 or ARM hardware platform, separation kernels can simultaneously support x86, ARM, SPARC, and other virtual machines. Admittedly, at times this must be done through emulation with substantial performance penalties but without requiring modification of

executable code including operating systems that would run natively on those hardware platforms. This can substantially reduce the costs of porting software and facilitate safe migration from aging legacy hardware platforms.

Also, because the software for enforcing separation is cleanly segregated from the software for providing virtualization, separation kernels naturally facilitate creation of many small and lightweight partitions where software may run at native speeds within the constraints and strong protection of the separation kernel but without the substantial overheads of virtualization. This is critical for efficiently providing strong separation and protection of down-graders, re-graders, cryptographic components, and components for mediating covert channels in communications with shared devices such as hard drives and network interface cards, and protecting those components from untrusted partitions of untrusted security domains which users must access while their machine also hosts such trusted components. In such an architecture based on separation kernels, those many components can run safely and relatively securely at native speeds without the overhead of virtualization. By contrast, in a hypervisor based architecture, each component would either be required to run with the overhead of virtualization where the burden of the virtualization vastly exceeds the burden of the component itself, or they would be required to run outside the hypervisor where they would run without benefiting from even the softer assurance of separation provided by the hypervisor.

Last, most hypervisors have little functionality for controlling communications between the "partitions" they separate. In contrast, because separation kernels provide strong separation among these partitions, bi-directional and uni-directional communications between partitions can be permitted through specification of simple policies of who is permitted to read from whom.

In summary, separation kernels not only provide stronger separation between security domains forced to coexist on the same hardware, but separation kernels also provide high assurance protection of other technologies critical for effectively protected and controlled collaboration between domains in very large networks. Moreover, separation kernels provide high assurance protection for these technologies even while easing their deployment. These other technologies include kernel integrity monitors; release review components; and well separated instances of strong software based cryptography, with each instance appropriately supporting a different single classification level (or specific directional pair of classification levels) and each instance appropriately protected from the untrusted software existing within that classification domain. Moreover, not only does the separation kernel facilitate stronger separation of so many "compartments," but it can also provide more strongly controlled communication between compartments in a manner evaluated far more thoroughly than can be evaluated in hypervisors. The formal methods necessary for such strong evaluation of separation properties simply do not scale to the size of monolithic hypervisors, and even if the verification methods scaled, they would simply better illuminate the tangling of separation and virtualization violating the separation properties. In contrast, with formally verified separation provided by the separation kernel, the set of properties that must be formally verified for each of the security critical protected components is vastly minimized, and reduced in some cases to a scale where the properties of the components themselves can be formally verified.

These advantages of separation kernels apply to strong separation between effectively permanent domains of intranets, extranets, safety critical systems, the open Internet, personal domains for finance, friends and family, and the many ad hoc domains necessary for managing national scale emergencies, and international collaboration managing natural disasters and transnational threats. Finally, strong separation kernels can provide such separation while providing the full functionality to each domain that each domain desires.

3.2. Hardware support of secure virtualization

Given these strengths of separation kernel based virtualization, why seek increasingly secure hardware? Although it is possible to do many things in software, software cannot effectively attest to the integrity of the underlying software and hardware environments. Trusted Computing Group standards [11] help address the limitation.

However, other challenges remain. For instance, Unfortunately, the x86 instruction set is not easily accurately virtualized.[12] Kernel code and device drivers generally have to operate in privileged mode ("ring 0") in order to execute the instructions that directly manipulate hardware, page table registers, etc. Historically, most virtualization software has run the guest OS code at user level, and trapped privileged instructions to emulate the requested activities. However, in order to run a guest OS unmodified, it is critical that the virtualization system be extremely faithful to the expected semantics of the privileged instructions and hardware. Any discrepancies can

cause the guest OS to crash badly. For instance, given that instructions which write to privileged registers trap as expected, but many of the instructions that read from a privileged register do not trap at all, a guest compartment might be writing changes to the virtualized page table base address but reading from the *actual* page table address, possibly causing it to fail spectacularly. In order to overcome this, virtualization systems have had to resort to other strategies such as inserting breakpoints into every piece of code that contains these problematic instructions. Then ensuring that the breakpoints are maintained consistently requires tremendous effort and resources. A complete discussion of these sorts of measures can be found in [13].

However, Intel Corporation has recently added new support for virtualization in the form of the Vanderpool Technology (VT) instructions included in the latest Intel Core processors which, simply put, provide a new execution mode (which can be thought of as "ring -1") beneath the previously most privileged level. This allows guest Operating Systems to run at level 0, completely unaware of the fact that they are running in a virtual environment. But any attempt to access *any* of the formerly problematic instructions will trap to the emulation code provided by the virtualization layer.

Hardware support for virtualization and motherboard support for system integrity monitoring are now rapidly becoming widely available in the mass quantities commercially feasible for use by average customers. Several leading providers of personal computing hardware already ship platforms with such technology. For instance, the Intel Corporation hardware-assisted virtualization technology known as VT is similar to AMD Pacifica Technology. Further, Intel Corporation secure platform technology known as LaGrande Technology (LT) [14] is based on TCG standards [11] and similar to AMD Presidio Technology. LT and VT already ship embedded together from leading providers of desktops and servers. Moreover, TCG compliant technologies already ship in desktops, laptops and servers of many leading providers.

3.3. Uses for secure virtualization

In addition to allowing users to safely and securely interact with multiple security domains practically simultaneously, and in addition to easing development and deployment of guards, secure virtualization has a variety of other important use-cases.

Currently, most host based security technologies depend on at least partial integrity of the underlying OS, and much of the underlying hardware. Such host based security technologies include antivirus, host based intrusion prevention, antispyware, and personal firewalls.

Hardware support for secure virtualization can narrow the hardware dependencies and eliminate dependence on the OS by allowing the security technology to exist in an independent partition simply watching, and not trusting, the monitored OS. In fact, the security software may even become the virtual machine monitor such that the untrusted OS depends on the security software without the security software depending on the untrusted OS.

Moreover, because separation kernels can protect components from tampering by other components, they facilitate effectively protected insertion of tamper resistant host integrity monitors, network intrusion prevention systems (N-IPS), and network intrusion detection systems (N-IDS). Additionally, separation kernels can force redirection of all interaction from any component bound for hardware to become an interaction with the appropriate mitigation component. In this way they provide non-bypassability to the N-IDS and N-IPS whose transparent and strongly protected integration they facilitate. Such N-IDS and N-IPS can be inserted between the network interface card (NIC) and the untrusted OS. Alternatively, separation kernels support protected and transparent insertion of such monitors between any Guest OS and any cryptographic software, and further ensure that the cryptography is not bypassed or tampered.

In these ways, separation kernels help protect and ease deployment and verification [17] of the mitigation components mediating communication of untrusted fully functional guest operating systems with shared hardware and the rest of the outside world. Worth noting, as separation kernels protect the supporting compartments from tampering, integration of LT may help separation kernels by ensuring secure boot and attestation.

3.4. Secure operating systems today

Of course, separation kernels are not new, and a variety of high assurance operating systems have served the community for decades, including Provably Secure Operating System (PSOS), GEMSOS, MULTICS, and others. However, the market for high assurance and real time operating systems has grown dramatically since creation of many of these operating systems. A number of high assurance and real time operating systems now have revenue streams from commercial applications in avionics and other embedded applications driving improvements in

performance and verifiable functionality. Although this does not change the realm of the possible, it does change, dramatically, the realm of the feasible. Further, with multi-billion dollar problems of identity theft and intellectual property theft climbing exponentially, the demand for separation properties has never been higher.

As an example of such a relatively secure separation kernel based high assurance real time operating system, Green Hills Software has developed a separation kernel that is currently being evaluated by a Common Criteria Testing Lab, SAIC, in Columbia Maryland for meeting the requirements of an Evaluation Assurance Level (EAL) of 6+ against the Separation Kernel Protection Profile (SKPP).[15] The formal methods artifacts for the EAL6+ evaluation were developed by Rockwell Collins Formal Methods Center of Excellence under the F/A-22 and F-35 programs under management of Air Force Research Labs. This separation kernel borrows heavily from a Real Time Operating System commercially developed by Green Hill Software. This separation kernel has been certified under eight different DO-178B Level A certifications for "fly by wire" real-time aircraft control and other flight-critical control functions. This separation kernel is also used by the Department of Defense in the Joint Tactical Radio System and Intel Corporation is projected to soon ship LT for mobile hardware. Also based on this separation kernel, Green Hills Software has developed a proof-of-concept multi-level secure workstation capable of running different operating systems in parallel with strong separation properties.

3.5. Limitations of separation kernels

Despite the strengths of separation kernels, by themselves, separation kernels are not sufficient to provide separation throughout the entirety of a typical workstation. Separation kernels provide high assurance separation of CPU behavior and memory interactions and enforce rigorous static allocations of timeslots of resources such as CPU time, bus bandwidth, and memory, to help ensure that resource contention signaling is kept to a minimum. However, typical workstations have peripherals that must be shared between security domains, including hard drives, hard drive controllers, a video card, network interface card, keyboard, mouse, and other Universal Serial Bus (USB) peripherals. Safely and securely sharing such resources requires careful development and integration of software to mitigate the overt and covert channels to and from such peripherals.

Toward mitigating the overt channels to and from storage and communications, all content can be encrypted. However, this leaves both covert channels to and from such peripherals, and also covert and overt channels to and from other peripherals such as keyboard, mouse, and video. Keyboards and mice are simpler cases since these devices simply relay user input keeping very little state over time, facilitating frequent, clean, and strong resets to trusted "clean" states. Additionally, it might be possible to encrypt all communications from such input devices to prevent eavesdropping by others. In many ways, even the covert channels to and from keyboard and mice might be easier to manage in that masking covert channels of a few hundred words per minute seems likely to be simpler than masking covert channels to and from near gigabit throughput hard drives bursting tracks off platters spinning at thousands of revolutions per minute.

Even if the overt channels can be fully encrypted, mitigating covert channels to and from high speed devices with either persistent state or strong capacity for resource contention signaling, such as hard drive controllers and network interface cards, requires substantial work. Specifically mitigating such covert channels requires integration of mitigation components to mask timing channels, sequencing channels, provide a layer of indirection in addressing, and mitigate other covert channels as well. Fortunately, since the separation kernel provides strong separation of security domains in processor and memory, such mitigation components can run "protected" in such a shared processor and shared memory while mitigating the covert channels to and from each peripheral.

It is important to note that video can be a special case. Specifically, trusted video hardware may be needed if untrusted security domains require the ability to read status or other feedback related to their section of a screen and (a) there is a requirement for video to simultaneously display cleartext data (text, image, video, etc.) of multiple domains or (b) the video card cannot be adequately reset after presentation of content from one domain and before presentation of content from another domain. Fortunately, LT includes hardware support for constructing trusted paths through display adapter hardware.

3.6. Mitigation Components

As mentioned above, even if all content is encrypted prior to being sent to a shared peripheral, the covert channels to and from such peripherals must be mitigated.[16] The potential approach mentioned in the section above involves integrating strongly protected and carefully built software components to mitigate covert channels. Even though such mitigation components can run protected by the separation kernel, separation and non-interference of information from different security domains must be maintained within the mitigation components for any mitigation component that interacts with multiple security domains. Maintaining this separation and non-interference is important for mitigation of covert channels even if the overt channels of content are encrypted before being read into the mitigation component.

In other words, the separation kernel protects each supporting compartment from interference or tampering by other compartments, dramatically easing formal proofs [17] that the supporting compartments are correctly built and able to perform their necessary functions. However, despite this help, the information flow properties within the mitigation component must still be verified. Interestingly, these proofs are still necessary to preserve strict control of covert channels even if the overt content is encrypted. Fortunately, separation kernels can provide the high assurance separation needed to safely and securely decompose large components into smaller components that are often small enough to have information flow formally verified. This often makes formal verification of mitigation components tractable but still non-trivial.

Also, experience in embedded systems has shown that even though each of the supporting compartments runs as an individual compartment for formally verified control of leakage, these supporting compartments have a negligible impact on CPU and other resource utilization of the guest computing compartments. This is because the supporting compartments contain only exceptionally small (trusted) components and do not require loading of virtualization or an OS atop the separation kernel.

However, given the challenges (and tractable but near Herculean effort) of verifying the separation properties of the separation kernel, and given the necessity of verifying specific properties of the mitigation components, further attention to formal verification techniques and their limits seems appropriate.

4. Formal Verification

Formal verification has advantages over less rigorous approaches to security evaluation. Formal verification is repeatable, independently verifiable, capable of consistent results independent of biases among evaluators and evaluation contexts, and can produce systems of much higher assurance. In contrast, most of the current "Common Criteria" approach to security evaluation depends upon correctness of manual generation and manual review of large volumes of text including security targets, objectives, configuration guidance, and threat environments that are susceptible to human error, susceptible to differing interpretations and differing judgement, and neither mechanically provably correct or formally verifiable. Malicious adversaries with billions of dollars of resources available to them to compromise computing systems seem capable of finding flaws that escape manual analysis but are revealed through mechanical analyses seeking to prove security or insecurity of a system. Such issues have long been a concern for organizations facing such adversaries. However, as organized crime masses billions upon billions of dollars of damage to legitimate businesses and unsuspecting consumers through identity theft and intellectual property theft, knowingly or unknowingly more parties face such adversaries every year.

Formal verification has other advantages over textual common criteria. Among these, formal verification includes proofs of completeness, and, if the theorems are well formed, formal checking can be done very quickly.

Admittedly, formal verification often requires large volumes of manually written and human readable text to explain the proofs and what is being proven. However, the proofs themselves can be mechanically verified for correctness.

Unfortunately however, fixing all issues revealed by formal verification to build a provably correct system is still a painstakingly slow process. Also, the process of carefully formulating the theorems needing proof can also be a painstakingly slow process. More importantly, technologies for formal verification do not yet scale to support verification of absence of all possible flaw types for systems as large as modern operating systems with Pentium class microprocessors and related chipsets. Verification of Pentium class chips is currently done piecewise with theorem proving technology derived from government research investments made decades ago, most commonly A Computational Logic and Applicative Common LISP

(ACL2).[18] However, just as the modular architecture of modern microprocessors facilitates piecewise formal verification of each module independently, the formally verified separation properties of a separation kernel facilitate formal verification of each of the higher level compartmented components individually.

4.1. Accelerating Evaluation

Given that developing a formally verified high assurance system such as an ACL2 verified system or an Common Criteria Evaluated Assurance Level (EAL) 7 system is a slow process, how much faster is the development of slightly lower assurance systems?

Obviously systems with much lower assurance, and systems without semi-formal, mechanical, common criteria or any thorough evaluation of any form are much faster to develop. So, a more important question seems to be, "How much slower is the development of formally verified systems relative to the development of slightly lower assurance systems?" Specifically, this could be asked in comparison of EAL 6+ or EAL 7 systems to EAL 4 or EAL 5 systems. The difference in cost and calendar time for evaluating otherwise similar systems at differing EAL levels is an average factor of two (2) for systems being evaluated at two levels apart with evaluation of an EAL 4 system averaging $250,000 over an average of sixteen (16) months, not counting development.[36] This practically guarantees that using a meaningfully evaluated system requires using systems that are outdated by a year or more and that effectively evaluating the many patches and updates for most current commercial and open source software is practically infeasible.

In this context, the deeper and more important questions seem to be, "why," and "to what degree can this be changed." EAL 4 and EAL 5 evaluations are considered expensive and time consuming because of the costs and delays associated with the manual nature of generating and reviewing the volumes of documentation. This manual process of generating and reviewing artifacts has helped give the current common criteria process a reputation for producing evaluation results that are both late and overly costly.

In contrast, formal verification is slow because formulating the theorems is a slow and careful process, and because fixing all of the issues revealed by formal verification requires painstakingly careful effort.

However, once the theorems are formulated properly specifying the properties desired to be verified as correct, actually mechanically proving or disproving the theorems is a relatively fast and relatively easily repeatable process.

This speed of verification is a particularly important point where the security critical requirements of a kernel or component do not change much if at all over time, even as the kernels and components themselves evolve. In this sense, once the overhead of establishing the framework for formal verification has been established, the assurance of a modified system can actually be verified much more quickly, not to mention much more reliably, through formal verification than through other verification techniques.

This seems counter-intuitive.

However, the micro-processor industry benefited directly from this advantage. Once the framework was established for formally verifying micro-processors, not only did the probability of a flaw escaping detection decrease, but the time required for verification of chips was decreased. Unfortunately however, although it might be possible to realize such advantages in separation kernels and smaller components, realizing such advantages on broader classes of software systems will require increasing scalability of formal verification techniques.

4.2. Challenges in scaling formal verification

Scaling represents the singular hard problem in applying theorem proving to modern software systems. Experiences show that: specifications, the scale of executable model, and the verification proofs all scale roughly linearly with the size of the system being evaluated, where "size" is often measured in a count of transistors or logical gates. However, as hardware has been growing exponentially as a function of transistor density in keeping with Moore's Law, software has been growing exponentially in response to the available computing power. To verify systems of the desired scale, much work must be done in (a) improving performance of model verification such that it completes in reasonable time for systems of the desired scale, and also in (b) creation of tools to facilitate more rapid specification of Theorems and Lemmas relevant to desired behavior.

It is important to note that it is not currently known if such improvements in scalability of formal verification can be achieved. However, each good inch of progress in scaling formal verification can increase the variety and types of software systems that can have their security critical properties verified to be correct, and also increases the scale of performance optimization that can be accomplished within formally verified components without increasing risk that such optimization violates desired security properties.

However, even if scalability of formal verification is increased dramatically, it seems likely that functional needs will always require software with functionality whose complexity is beyond formal verification, regardless of whether the functional needs are operational, mission, business, or consumer needs. In such cases, it might make sense to bound the risks of using such software as tightly as possible through tightly constraining such software in a segregated compartment. However, mission, business, and consumer needs may also require a number of such software packages to work together in a single security domain. In such cases, it could be very valuable to understand the types and degrees of risk each software package imparts on the others.

5. Defect Density

To estimate the types and degrees of risk each software package imparts on other software packages in such cases, we propose leveraging emerging techniques in model checking and binary analysis to improve software defect density estimation.[19] The closest analogy in other fields of engineering seems to be that it is not always possible or feasible to effectively non-destructively test and verify the quality of each cubic inch of all materials going into a large physical construction. However, some materials have an acknowledged probability of a defect that might only demonstrate itself in extended operation. When such probabilities are known, balanced construction of higher reliability structures is possible. In software, the pre-release defect density predicted from static analysis and the actual pre-release defect density are strongly correlated at a high degree of statistical significance.[37] Using such analyses to estimate risks several advantages, limitations, and dangers when applied to software and computer security.

Analyzing software to estimate the number of residual flaws, bugs or vulnerabilities has several advantages. First and foremost, such information is useful in assessing relative risk in deploying new code into sensitive environments. Second, if all of the known residual vulnerabilities are secretively monitored through techniques such as vulnerability-specific and exploit agnostic monitoring techniques, [20, 21] then having an estimate of the number of unknown residual vulnerabilities provides a measure of confidence that if there is an attempt to exploit a residual vulnerability that the attempt might be detected.

However, there are several dangers in oversimplifying the "measure" of confidence toward a "probability." Primarily, unlike natural faults which have a natural distribution, the malicious behavior of more sophisticated and better resourced adversaries is often biased toward the vulnerabilities which are more difficult to discover.

5.1. Steps toward estimating defect densities

Tools for the detection of vulnerabilities in code have improved significantly from finding "no" vulnerabilities at livable false positive rates to detecting substantial fractions of vulnerabilities at livable false positive rates. However, at the same time software is continuing to grow in both scale and complexity. Overall, software defect detection has improved tremendously from code reading and functional testing.[22]

With progress of model checking [23, 24] and other areas of static and dynamic software analysis, tools such as those distributed by Coverity and Fortify now scale to effectively analyzing millions and tens of millions of lines of code. This progress also includes lowering false positives from hundreds of false positives per true positive to a level where false positives and true positives are roughly equal while dramatically increasing the number of bugs found and breadth of types of bugs found. Experience annually evaluating several competing tools has shown that the best current tools now find about 30% of the inherent vulnerabilities while reporting only one false positive per true positive, even in scaling to systems involving millions of lines of code, and are capable of enabling a single person to effectively analyze millions of lines of code in a single week.

Unfortunately however, the performance described above is performance in analyzing source code, and the process of translating source code into machine-code level instructions of executable code can introduce devastating faults. For such reasons, an ability to perform effective model checking on practical binaries could be very important.

Fortunately, there has been progress toward model checking on practical binaries. The general approach is to extract an "intermediate representation" (IR) from the executable code. Such IR are neither source code, nor executable code. Instead, such representations attempt to infer the source-code constructs which originated specific patterns in the executable code. However, such constructs can be represented while preserving the precise underlying machine code mechanics of the implementation without any of the "loss of fidelity" associated with generalizing to source code. In this context, effective and scalable IR

extraction are needed, and model checkers need to be ported to run atop extracted IR.

There has been tremendous progress in extracting IR from binaries through several techniques, including Aggregate Structure Identification, Affine-Relation Analysis, and Value Set Analysis.[25, 26, 27, 28, 29]. Without sacrificing fidelity, the completeness of IR extraction is slowly growing toward a completeness suitable for model checking. However, none of the existing highest fidelity IR extraction techniques scale to the size of programs that can be analyzed with leading source analysis tools.

This is particularly unfortunate in that model checking on binaries can not only find bugs not appearing in source code, but analyzing binaries can also determine which reported "bugs" were in fact optimized away by the compiler.[30] Analyzing binaries also has several other advantages as well, including: [30]

- Ability to analyze code when source is not available,
- Verifying assumptions such as ANSI-C compliance,
- Analyzing compiler and post-compiler optimizations, and
- Analyzing inline inclusions of assembly code.

For these reasons, simply analyzing the source code seems insufficient for gauging the trustworthiness of a compiled program. In this context, the desired progress in improving completeness of IR extraction and improving scalability of IR extraction techniques as described above seems very important. If such "model checking of binaries" could be made possible, then it might be possible to substantially improve accuracy in estimating the number of unfixed flaws or bugs within a given binary. Given the rate of progress in this area, it seems possible and perhaps even likely that such goals will be reached in research within a few years, assuming government sponsorship of such research continues. In interim, various forms of static analysis in source code including model checking and taint analysis can be used to enhance testing of executable binaries.[31] However, given that testing and dynamic analysis do not necessarily provide full coverage, or always fully explore the corner cases more likely to be error prone, integrating such model checking into the binary analysis suite still seems valuable.

It seems unlikely that any tool will be able to find all vulnerabilities in software of size and complexity beyond formal verification with a thoroughness of theorem provers such as ACL2. For this reason, the best that should be expected from analysis of such software is an estimate of the number of residual bugs, flaws, and vulnerabilities in the software remaining to be fixed, and an incomplete list of specific potential bugs, flaws, and vulnerabilities that might need to be fixed. Over time, applying analysis tools and techniques to well studied code bases with a number of bugs, flaws, and vulnerabilities discovered through other techniques, it should be possible to begin estimating the false negative rates of such tools and analytic techniques along with ratios of false negatives to true positives and false positives across ranges of conditions. This facilitates better estimating the number of residual flaws, bugs, and vulnerabilities in new code on first analysis, even if the specific false negatives cannot yet be identified in the new code.

Such techniques for quickly applying a set of tools to a previously unseen piece of software to estimate the degree to which it might be safe to trust the software provide a potentially complimentary alternative to trusting the developer's estimates of defect density. However, where developer's may be trusted and their accuracy verified over time, techniques of process measurement can be very helpful where the developers track the time and phase that each defect is found and project forward the number of defects left to be found in the future.[32]

6. Transitive Risks

Systems have edges and interfaces. Even with strong separation of separation kernels and cryptographic separation, most security domains must interface other security domains. With risk measured so carefully within an isolated domain, how can we begin to measure risk for systems that are isolated except for a finite number of closely studied interfaces?

We discuss both human-machine interfaces and interfaces between security domains. We begin with human-machine-interfaces since the cases of wittingly or unwittingly allowing a malicious adversary direct logical, physical, and/or lifecycle access to a system provide points of origin for coordinated or uncoordinated malice, even if layers of protective domains exist between origin and potential targets of malice. Given the challenges of lifecycle access, critical unsolved aspects of lifecycle access are discussed last.

Most large scale systems at least have an interface to a number of people. Where more than one person shares a system, the motives and capabilities of other people are never certain. Psychologists and counter intelligence staff may reason on motives, and malicious capabilities are only rarely known with high certainty. However, worst case estimates of

capabilities may be useful in developing conservative estimates of a system's resilience to malice.

Each user may access a surface of the system and most users may inject a volume of data into the system, extract a volume of data from the system or both. The injection may be monitored and may even be filtered. Whenever a party has access to a shared security domain, the risk to the other party is practically unbounded, unless all input can be effectively filtered.

If the appropriate properties can be proven for a system's handling of input, then it might be sufficient to fully consider the input effectively filtered. However, for systems of a scale and complexity where such proofs are not possible, it may be appropriate and necessary to estimate whether or not there exists a defect that is reachable by the input. Such estimates might depend heavily on defect density estimation, size and complexity of input types to be received, and estimates of the amount of code that might be reachable by the input where such estimates could be done through dataflow analysis [33], slicing, [34], or improved forms of slicing

Vulnerabilities in jpeg rendering and libraries for simply playing audio files represent excellent examples of how incredibly common data types can be sufficiently complex for real trouble. As a more extreme example, where the surface, volume, and filtering of a party's access to a shared system permits them to introduce arbitrary code without strong separation of the code from the other party's interests, it would be conservative for the other parties to consider the domain compromised. Injecting arbitrary code into a shared domain may be as simple as the unrestricted ability to copy a file from a physical interface that accepts memory sticks, or a sufficiently large and ineffectively checked entry via a logical interface into an electronic form.

However, with the advantages of secure virtualization based on increasingly secure hardware, the size of a domain may be a small share of a single physical machine, span many physical machines, span several shares of a single machine, or span several shares of several machines. In each case, if there is not strong separation of one party's code from another party's interests, it is reasonable to suspect the domain might be compromised.

This brings us to interfaces between security domains. As with human-machine interfaces, interfaces between security domains have a surface, permit a volume of content to flow, and a security domain may attempt to process input to effectively filter anything potentially malicious. As above, if input is permitted to arrive via the interface without effective checking, the domain should be considered compromised by the input. As above, "effective" is defined as either fully proven to be effective, or estimated through estimates of whether or not there might exist a defect that might be reachable by the input.

As above, permitting injection of arbitrary code should be considered to compromise the security domain. By "code" we mean "executable instruction." In formally verified systems it might be possible to strongly differentiate between code of the system that is immalleable and proven to behave properly for all data, and the data which is processed by the code and proven to not be permitted to change behavior of the code or configuration of the environment in any risky ways. In other cases, we begin down the slippery slope of analyzing the range of data that might be received across interfaces, and estimating the potential for some input to reach a defect. If such input is able to directly or indirectly reach a defect, such defects might permit either triggering of instructions, translation of data into executable code, or changing behavior of the code or configuration of the environment in risky ways. Any calculus for extracting probabilities or risks from such a model should recognize that well resourced adversaries study a system until they find a suitable vulnerability. In such cases, the question is not the probability of a random adversary finding a vulnerability, but rather the question of "what level of resources are necessary to find such vulnerabilities."

In that context, attempts to increase verifiability of systems then drives systems architectures toward sequences and meshes of much smaller domains with strongly controlled interfaces. These domains might often be as small as possible to at least have the estimated risk minimized and at best have their properties effectively provable. For example, perhaps a single physical server might have one domain storing a set of web pages which the server-daemon domains may only read. Perhaps only the author's clearly separated domain would have permissions to modify the page storage domain. To ensure that no user interferes with a server-daemon serving another user, a new server-daemon domain and new server-daemon could be instantiated for each new address visiting the server, each server-daemon domain could spawn input processing domains for filtering data submitted via forms interfaces, and the filtering of any input could be done through a series of domains structured for decomposing and analyzing the input with each domain small enough to have provable properties. Filtered data could be read into transaction processing domains, and resource utilization could be monitored from a transaction management domain with all of the

protected components and interfaces small enough and simple enough to have provable properties.

The performance impact of fragmenting a server into such "micro-domains" and instantiating such redundancy are likely impractical for most if not all applications today. However, with the cost of annual damages from identity theft and intellectual property leakage climbing exponentially, the number of life critical applications of computing climbing exponentially, system performance climbing exponentially, and the value of transactions entrusted to computing likely to eventually near "the entire economy," this may not always be the case.

6.1. Lifecycle Access and Transitive Risks

In the scheme described above, arbitrary code is greatly distrusted unless its properties are proven, and interfaces are closely monitored for anything with sufficient complexity as to introduce risks. However, teams of people routinely write large volumes of code in scale well beyond formal verification, and this code is injected routinely into many organizations around the world. Subtle bugs planted by malicious developers evade defect detection, [35] and statistics on defects do not accurately capture the harm that can be done if there is collusion between a malicious developer and a party with direct or indirect access at run time. This may drive development of some types of applications toward formally verifiable scales, but it is uncertain that all critical applications can ever be reduced to such scale, even if there is progress in scalability of formal verification. Perhaps it might always be the case that if you buy something incredibly complex from someone wanting to hurt you, you might get hurt, badly. Perhaps some problems are beyond technology. However, when I consider the range of woes that can be stopped through stronger separation of information at rest, in transit, and in processing, when I consider the range of woes that can be addressed by beginning to quantify and reduce risk in software, and when I see such progress in binary analysis, separation kernels, and hardware support for secure virtualization, I grow optimistic that we might see a path emerging that might take us a long way toward safer and more secure computing.

7. Engineering Sufficient Security

Best practices currently involve adding security technology into or around computing systems designed to function with or without security technology. Examples of such security technology include firewalls, intrusion detection, antivirus, antispyware, and antispam. However, it is difficult to measure the security of the resulting architecture, and difficult to measure whether risk is increasing or decreasing as complexity of the network and sophistication of defenses increase somewhat in parallel. Also, it is difficult to make claims much beyond protection of known vulnerabilities.

An alternative approach might be to create a foundation which can provide provably strong security where all information is encrypted in transit or storage, and protected by provably correct enforcement of separation properties whenever exposed for processing. In such an architecture, where information must cross from one security domain to another, it might be valuable to have a measure of the risks the information was exposed to while being processed in a cleanly separated compartment. These risks include the people and potential software bugs, flaws, and vulnerabilities that the information was exposed to, and through various static and dynamic analysis techniques which are continually improving, it should be possible to have effective measures of such risks. With measurable risk and such strong and flexible means of bounding risk, it becomes possible to develop coherent risk reduction plans.

The level of security needed may vary greatly with value or sensitivity of information. However, as consumers face identity theft, and businesses face losses of billions of dollars, it seems that such measures of risk and such a strong foundation might find broad usage.

8. Acknowledgements

I would like to thank Darrell Kienzle of Symantec Research Labs, Nic Watson of Green Hills Software, and Warren Hunt of the University of Texas at Austin for their substantial and invaluable contributions to this paper. I would also like to thank Carey Nachenberg, Tom Haigh, Darren Shou, and Chris Wysopal for their critique, suggestions, and very helpful recommendations. The best parts are theirs, and the rest are mine.

9. References

[1] John Rushby, Kernels for Safety? in Safe and Secure Computing Systems, chapter 13, pages 210–220. T. Anderson, editor, Blackwell Scientific Publications, 1989.

[2] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, D. Boneh, Terra: A Virtual Machine-Based Platform for Trusted

Computing, Proceedings of the 19th Symposium on Operating System Principles, October 2003. Proceedings of the nineteenth ACM Symposium on Operating systems principles, pp. 193-206.

[3]Meushaw , et al., Device for and method of secure computing using virtual machines, United States Patent 6,922,774, Awarded July 26, 2005, Assignee: The United States of America as represented by the National Security Agency (Washington, DC), Application No.: 09/854,818 Filed: May 14, 2001

[4] NetTop: Technology Profile Fact Sheet, http://www.nsa.gov/techtrans/techt00011.cfm

[5] INTEGRITY PC Secure Virtualization Solution for Linux and Legacy Applications, http://www.ghs.com/products/rtos/integrity_pc.html

[6] U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness, Version 1.0. Information Assurance Directorate, National Security Agency

[7] D. Greve, M. Wilding, and W. M. Vanfleet, "A Separation Kernel Formal Security Policy", Fourth International Workshop on the ACL2 Prover and Its Applications (ACL2-2003), Boulder, CO, July 2003.

[8] P. Barham, B. Dragovic, K. Fraser, S.Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfied, Xen and the Art of Virtualization, Proceedings of the 19th Symposium on Operating System Principles, October 2003.

[9] VMM Software Architecture Options, http://www.intel.com/technology/itj/2006/v10i3/2-io/3-vmm-software-architecture.htm

[10] Intel Virtualization Technology for Directed I/O Architecture Specification, Intel Corporation, D51397-001, February 2006.

[11] Trusted Platform Module Main. Part 1 Design Principles, Specification Version 1.2, Revision 94, Trusted Computing Group, March 2006.

[12] J. S. Robin and C. E. Irvine, Analysis of the Intel Pentiums Ability to Support a Secure Virtual Machine Monitor, Proceedings of the 9th Usenix Security Symposium, XP002247347, Denver, Colorado, pp. 129–144, Aug. 14, 2000.

[13] Kevin Lawton, Running multiple operating systems concurrently on an IA32 PC using virtualization techniques available at http://www.floobydust.com/virtualization/lawton_1999.txt

[14] LaGrande Technology Preliminary Architecture Specification, Intel Corporation, May 2006.

[15] INTEGRITY-178B Separation Kernel Security Target, Green Hills Software, 2005.

[16] Partitioning Communication System Protection Profile, Objective Interface Systems, 2005.

[17] J. Rushby, Design and Verification of Secure systems, Proceedings of the 8 th ACM Symposium on Operating System Principles, Pacific Grove, California, 14-16 December 1981.

[18] Matt Kaufmann, J. S. Moore, ACL2: An Industrial Strength Version of Nqthm, Compass'96: Eleventh Annual Conference on Computer Assurance, 1996.

[19] Security Vulnerabilities in Software Systems: A Quantitative Perspective, O. H. Alhazmi, Y. K. Malaiya and I. Ray, Colorado State University, http://www.cs.colostate.edu/~malaiya/635/IFIP-10.pdf#search=%22%22software%20defect%20density%20estimation%22%22

[20] Carey Nachenberg, Generic Exploit Blocking, Virus Bulletin, February, 2005.

[21] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier,,Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits In the Proceedings of ACM SIGCOMM, August, 2004, Portland, OR.

[22] C. M. Lott and H. D. Rombach, Repeatable software engineering experiments for comparing defect-detection techniques, Journal of Empirical Software Engineering, 1(3), 1996.

[23] Engler, D., D.Y. Chen, S. Hallem, A. Chou, and B. Chelf.. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code, In SOSP 2001. 2001.

[24] Chen, H., D. Dean, and D. Wagner, Model Checking One Million Lines of C Code, In Symp. on Network and Distributed Systems Security (NDSS), 2004.

[25] Ramalingam, G., J. Field, and F. Tip, Aggregate Structure Identification and Its Application to Program Analysis, In POPL 1999.

[26] M. Karr, Affine relationships among variables of a program, Acta Informatica, 6:133--151, 1976.

[27] Müller-Olm, M. and H. Seidl, Interprocedural Analysis of Modular Arithmetic, in ESOP. 2005.

[28] Müller-Olm, M., H. Seidl, and B. Steffen, Interprocedural Analysis (Almost) For Free, in ESOP. 2005.

[29] Codesurfer/x86-a platform for analyzing x86 executables, in R. Bodk, editor, CC, volume 3443 of Lecture Notes in Computer Science, pages 250--254.

[30] Reps, T., Balakrishnan, G., and Lim, J, Intermediate-representation recovery from low-level code, in Proc. Workshop on Partial Evaluation and Program Manipulation (PEPM), (Charleston, SC, Jan. 9-10, 2006).

[31] Michael D. Ernst, Static and dynamic analysis: Synergy and duality, in WODA 2003: ICSE Workshop on Dynamic Analysis, (Portland, OR), May 9, 2003, pp. 24-27.

[32]Ebert, C., Dumke, R., Bundschuh, M., Schmietendorf, A., Best Practices in Software Measurement, How to use metrics to improve project and process performance; 2005, XII, , Hardcover ISBN: 3-540-20867-4 .

[33] Reps, T., Horwitz, S., and Sagiv, M., Precise interprocedural dataflow analysis via graph reachability. In Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages, (San Francisco, CA, Jan. 23-25, 1995), pp. 49-61.

[34] Mark Weiser. "Program slicing," IEEE Transactions on Software Engineering, vol. SE-10, no. 4, July 1984.

[35]Ken Thompson, Reflections on Trusting Trust, Communication of the ACM, Vol. 27, No. 8, August 1984, pp. 761-763.

[36] United States (US) Government Accountability Office (GAO) report GAO-06-392, “INFORMATION ASSURANCE: National Partnership Offers Benefits, but Faces Considerable Challenges,” March, 2006.

[37] Nachiappan Nagappan, and Thomas Ball, Static Analysis Tools as Early Indicators of Pre-Release Defect Density, Proceedings of the 27th international conference on Software engineering, St. Louis, MO, USA, Pages: 580 – 586, May, 2005, ISBN:1-59593-963-2