Dancing with the Adversary: A Tale of Wimps and Giants

Virgil Gligor(✉)

Carnegie Mellon University, Pittsburgh, PA 15213, USA
gligor@cmu.edu

Abstract. The long-standing requirement that system and network designs must include accurate and complete adversary definitions from inception remains unmet on commodity platforms; e.g., on commodity operating systems, network protocols, and applications. A way to provide such definitions is to (1) partition commodity software into "wimps" (i.e., small software components with rather limited function and high-assurance security properties) and "giants" (i.e., large commodity software systems, with low/no assurance of security); and (2) limit the obligation of definining the adversary to wimps while realistically assuming that the giants are adversary controlled. We provide a structure for accurate and complete adversary definitions that yields basic security properties and metrics for wimps. Then we argue that wimps must collaborate ("dance") with giants, namely compose with adversary code across protection interfaces, and illustrate some of the salient features of the wimp-giant composition. We extend the wimp-giant metaphor to security protocols in networks of humans and computers where compelling services, possibly under the control of an adversary, are offered to unsuspecting users. Although these protocols have safe states whereby a participant can establish temporary beliefs in the adversary's trustworthiness, reasoning about such states requires techniques from other fields, such as behavioral economics, rather than traditional security and cryptography.

1 Introduction

A system without accurate and complete adversary definition cannot possibly be insecure. Without such definitions, (in)security cannot be measured, risks of use cannot be accurately quantified, and recovery from penetration events cannot have lasting value. Conversely, accurate and complete definitions can help deny any attack advantage of an adversary over a system defender. At least in principle, the seemingly inevitable adversary-defender asymmetry can be reduced and secure system operation achieved. Hence, it seems important to design systems and networks that include such definitions from inception. However, this is unlikely to happen for commodity systems: although security has been recognized to be a fundamental problem, it has always been of secondary importance in the design of commodity systems, and it is very likely to remain that way;

B. Christianson et al. (Eds.): Security Protocols 2014, LNCS 8809, pp. 100–115, 2014.
DOI: 10.1007/978-3-319-12400-1_11

viz., the "axioms" of insecurity [6]. Nevertheless, this fact does not remove the obligation to provide accurate and complete adversary definitions. Adding secure components to insecure commodity systems or networks will continue to mandate it.

Although an adversary's attack advantage cannot be eliminated in large, low-assurance commodity software (i.e., for "giants" [6,12]), it can be rendered ineffective for small software components with rather limited function and high-assurance security properties, which are isolated from giants; i.e., for "wimps." However, isolation cannot guarantee wimps' survival in competitive markets, since wimps trade basic system services to achieve small attack surfaces, diminish adversary capabilities, and weaken attack strategies. To survive, secure wimps must use services of, or compose with, adversary-controlled giants.

In this paper, we propose a structure for adversary definitions that is consistent with those found in other areas of security (i.e., cryptographic schemes, or modes). The proposed structure is desirable. It can yield accurate and complete adversary definitions – just as it does in cryptography – and it is easily adapted for different wimp interfaces, ranging from cryptographic schemes, operating systems, application modules, and human protocols. We argue that accurate and complete definitions yield security properties and metrics, which are useful for the design of wimps. Then we explain why wimps must compose (i.e., "dance") with giants thereby illustrating the paradoxical theme of this workshop, namely the collaboration with the adversary. Finally, we extend the wimp-giant composition metaphor to security protocols in networks of humans and computers where compelling services, possibly under the control of an adversary, are offered to unsuspecting users. These protocols produce value for participants who collaborate. However, they allow malicious participants to harm honest ones and corrupt their systems by employing deception and scams. Yet these protocols have safe states whereby a participant can establish beliefs in the adversary's (perhaps temporary) honesty. However, reasoning about such states requires basic results from other fields, such as behavioral economics, rather than traditional security and cryptography.

2 Accurate and Complete Adversary Definitions for Wimps

Adversary-Controlled Giants. An adversary can be thought of as a program that launches a set of attacks at a system interface under the control of various input commands issued by humans. This implies that, to define an adversary accurately and completely, one must find all adversary-accessible interfaces of all system components, ranging from operating systems and network protocols to all system applications. Then, for each component, one must find all vulnerabilities that could be exploited by an adversary, and all attack strategies to exploit each vulnerability. Furthermore, an adversary could exploit different types of attacks against multiple components of a giant, and thus one must be able to account

for all possible attack combinations to obtain an accurate and complete definition; e.g., compose all attack capabilities and strategies. Since giants comprise hundreds of thousands of component interfaces of different sizes and complexity, and tens of million lines of source code, it is highly unlikely that an accurate and complete definition of an adversary will ever be possible. To make things worse, some giant code and interfaces may change faster than the time necessary to complete an accurate adversary definition for it. In short, one can safely assume that *a giant is always part of the adversary definition*. Hence, the only system components that could possibly be defended from adversaries are wimps. Thus, the obligation to provide accurate and complete adversary definitions can be limited to wimps. However, since wimps can be part of different system components, they can have vastly different semantics and thus one needs a fairly general and uniform structure for adversary definitions, since these definitions must compose.

Adversaries in Cryptographic Schemes. A security sub-field that has produced accurate and complete adversary definitions successfully for relatively small modules with precisely specified functions (i.e., wimps) has been cryptography. Although fairly coarse, this analogy is intended to make two points: (1) the structure of the adversary definition in cryptographic schemes serves as a good starting point, given that these definitions have been successfully used in proving properties of encryption/authentication schemes [19]; (2) just as in cryptography, where the adversary definition is part of a cryptographic scheme's specification, the adversary definition can be part of any wimp specification; i.e., for software modules of similar size and complexity.

Security of *cryptographic schemes* is defined in terms of an *attack game*, and a model of *adversary power and privilege*. An adversary can be viewed as the set of possible attacks that can be launched against the *scheme*. Informally, each attack consists of a triple: an adversary's *goal*, set of *capabilities*, and *strategies* that exploit capabilities to reach the goal. In encryption schemes, the scheme's interface comprises an encryption, and possibly a decryption oracle, and the goal may be distinguishability of ciphertexts leading to leakage of plaintext information. In authentication schemes, the interface is to an authentication-tag generation and a verification oracle, and the adversary's goal is to forge a plaintext or an authentication tag that passes the verification-oracle's check. Capabilities represent the adversary's ability to obtain verifiable, predictable, known, or chosen plaintext from the system or network – as needed – and invoke an oracle. Attack strategies include launching adaptive, interactive, or concurrent attacks; e.g., exercising both choices of plaintext and ciphertext to break plaintext secrecy or create ciphertext forgeries.

The adversary *power* (e.g., polynomially bounded/unbounded, deterministic/randomized program, types of operations and their speed and storage requirements) and *privileges* (e.g., access to an oracle's entry points, ability to selectively specify input data, and invoke a single oracle or more) specify how an adversary plays the attack game; e.g., whether the adversary can exercise a particular game strategy. Capturing *all* attack strategies is important because otherwise one can-

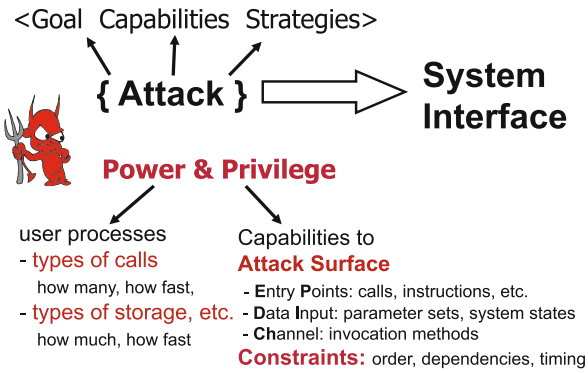

Fig. 1. Simple attack-definition template

not produce encryption or authentication schemes that demonstrably counter all attacks; e.g., provably support indistinguishability properties in encryption or unforgeability properties in authentication for different types of attack capabilities.

Structure of Wimp Adversaries. A similar adversary structure as that used for cryptography schemes can be applied to other types of wimps. As in cryptography, the adversary can be defined as the set of all possible attacks that can be launched at a wimp interface. In addition to the typical call interface, a wimp's interface must account for all sources of input; e.g., memory state, I/O devices, initial system state. As in cryptography, the adversary is a program, or a set of programs, that executes instructions based on inputs it receives from its users and/or other attack programs. However, the goals and capabilities of an attack game will be different, and so will the strategies. Nevertheless, just like in cryptography, we can define the attack game via <goals, capabilities, strategies> triples at different wimp interfaces. We also define the adversary's computationally bounded power and privileges in an analogous manner. For example, the adversary power includes a specification of how many end-hosts and processes operate the attack, how fast processors need to be, how much and what type of storage areas are needed, and what types of communication media and how much bandwidth are required. The adversary's power would have to be polynomially bounded – just as is done in complexity-based cryptography – since these wimps may use cryptographic schemes whose adversary is assumed to be bounded.

Figure 1 illustrates the template for an attack definition whereas Fig. 2 summarizes the use of the template with two attack examples. The size and complexity of the Xenix Kernel are not intended to approximate those of a wimp and are used only for illustrative purposes. In the attack of Fig. 2(a), the *attack goal* is to invoke the internal function *panic* of the Xenix operating system kernel [8] via unprivileged system calls and crash the system repeatedly, thereby causing persistent denial of service for system users. The *attack capabilities* comprise

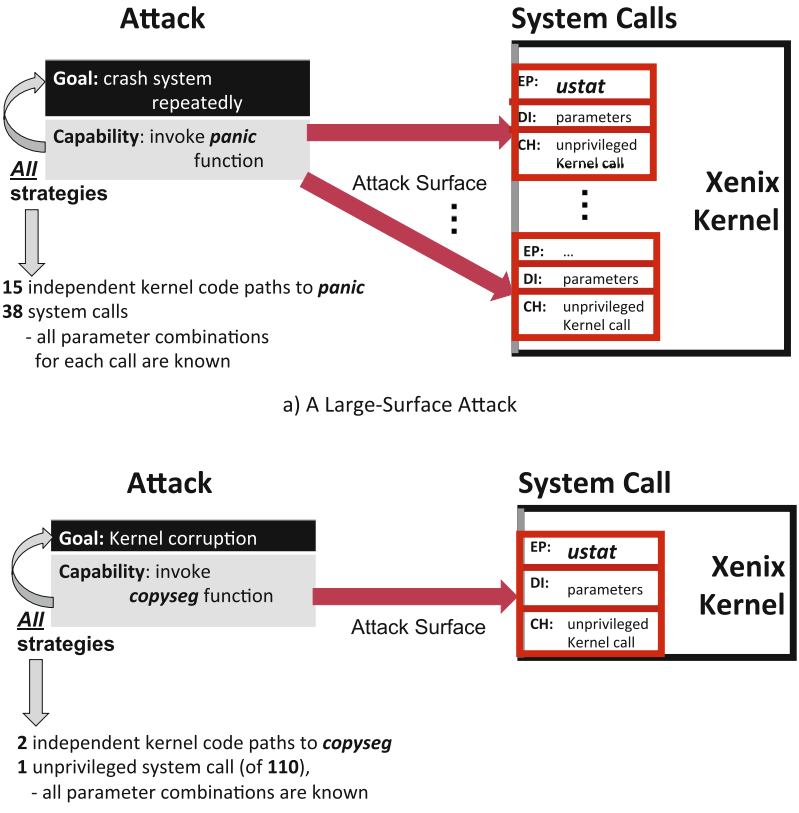

Fig. 2. Attack examples

access to 38 of 110 system calls and the *attack strategies* are all call-parameter combinations that trigger a crash. In a practical sense, the adversary's capabilities and strategies represent a measure of an attack's surface [9,15].

The power and privileges of the adversary are fairly common; i.e., the adversary is an unprivileged user-level program that invokes unprivileged kernel calls. In this example, there is no call ordering, timing or dependency constraint on call capabilities that are left unknown after source-code analysis of the kernel. This requires both control-flow analysis to identify all the 15 independent flow paths that lead to the *panic* function, and information-flow analysis to identify all call-parameter values and combinations thereof that activate these flow paths. In this example, the integrated (control and information) flow analysis reveals *all* strategies (i.e., kernel calls and call-parameter combinations) the adversary can employ to crash the system repeatedly. At the time of this analysis, the security properties of the Xenix kernel did not counter the activation of any of the 15 independent code paths that led to the panic function invocation.

In contrast with Fig. 2(a), which illustrates a large-surface attack, Fig. 2(b) illustrates an attack that has a small surface, which can be easily countered in practice, once discovered. The *attack goal* is to invoke the internal function *copyseg* of the Xenix kernel via an unprivileged system call with a parameter combination that enables overwriting adversary-selected values in kernel space, thereby corrupting kernel operation. The *attack capabilities* comprise access to a single system call and the *attack strategies* all call-parameter combinations that cause a kernel overwrite. The adversary is an unprivileged user-level program that invokes unprivileged kernel calls. Integrated flow analysis on source code indicates that there are two independent flow paths to *copyseg*, and thus the strategy space (i.e., kernel calls and call-parameter combinations) is limited, although additional flow-path activations are possible using privileged calls (which Fig. 2 omits). At the time of this analysis, the security properties of the Xenix kernel did not counter the activation of any of the two independent code paths.

Examples of small attack surfaces whose exploitation is via *probabilistic strategies* also exist. Typical examples are the so-called *time-of-check-to-time-of-use attacks*, which attempt to exploit specific vulnerable time windows in system implementation; e.g., the *binmail* attack [2] where the goal of the adversary is to get root privilege using a strategy that exploits a small time window in file (un)linking. This attack uses conventional capabilities and unprivileged system call invocation.

Attack Composition. To obtain a desired capability, an attack A may require a capability provided by meeting the goal of attack B, and this leads to the notion of *attack composition*. Attack composition requires that (1) the adversary capabilities, power, and permissions necessary for attack B do not conflict with (e.g., exclude) the other capabilities needed by A, and (2) the strategies used by B do not conflict with those of A. For example, in many business applications, if the success of attack B requires a capability to access an accounts payable application, then adversary launching attack A cannot obtain a capability for issuing purchase orders. Or, if launching attack B requires root permissions, then launching a successful attack A from the unprivileged mode is ruled out. Also, if the strategy employed by attack A's requires timely program execution, executing attack B's strategy must exclude crashing the system.

In attack composition, the goal of B may represent a capability needed by multiple attacks – not only by A – and this leads to (directed) *attack graphs*. That is, a node of an attack graph comprises the triple <goals, capabilities, strategies>, adversary power and privileges, and an edge connects the goals of descendant nodes to the capabilities needed by their ancestors.

We note that, even when instances of attack graphs are (directed) trees, these trees are different from those often illustrated for the past two decades [1,16,24,25], in at least three ways. First, each node defines the attack game, adversary power, and privileges, and hence it captures *all* attack execution strategies and capabilities needed, including their ordering, dependencies, and timing. Second, adversary privileges include a security boundary, or attack surface specification,

whose size and complexity is minimized by wimp definitions; e.g., all entry points required and input parameter combinations to exploit capabilities. Third, attack nodes offer a direct way to measure security strength, as explained below.

Our notion of the attack graph also differs from that of the more recently defined but more limited notion of the "kill chain" [10]. While the reconnaissance, weaponization, and delivery steps of a kill chain correspond to the notion of a set of capabilities to an attack surface discovery (i.e., entry-point, malicious data input, and delivery channel), the exploitation, installation and execution steps capture only a single attack strategy, instead of *all* strategies as required by our attack node.

We stress again that our attack structure is intended for the accurate and complete definition of wimp adversaries. Even if this structure may be applicable to giants in principle, it is unlikely that such definitions can be used in practice due to giants' inherent size and complexity.

3 Wimp Security Properties and Metrics

Accurate and complete attack definitions imply that a defender can design security properties to counter those attacks, and implicitly deny the adversary's (asymmetric) advantage. Some properties may deny certain attack strategies and/or capabilities and hence the adversary cannot reach his/her goal. Other properties may deter the adversary from using specific strategies or capabilities; e.g., by audit, by increased workload. Yet others may limit the attack's success; e.g., the defender may recover secure system states thereby forcing the adversary to retry the attack (and eventually get discovered); or undo the effects of an attack that corrupts system memory states.

Since an adversary attack comprises a program executing in response to user input commands, an adversary's *attack behavior* can be viewed as sets of instruction-execution traces. Attack behaviors can be countered by defining wimp interfaces, which restrict or block some execution traces whenever the adversary attack invokes a wimp. Hence, just as in cryptography, the adversaries' attack behavior becomes part of a wimp's definition, and a wimp's security properties become *negations* of adversary attacks. For notational simplicity, we denote the set of properties that counter an attack A by $\overline{A}$.

In turn, security properties can yield basic metrics of security. For example, we say that attack $A \Longrightarrow$ attack B if all security properties that counter attack B also counter attack A. For example, this relation is required when attacks A and B compose, and is strictly weaker than composition. Like composition, it is reflexive, anti-symmetric, and transitive. Attack A $\not\Longrightarrow$ attack B if not all security properties that counter attack B counter attack A. This relation captures cases when attack B can be used by, but is not necessary for, attack A or when the two attacks do not compose. Using these relations we can then define the notions of attack "dominance ($>$)," "equivalence" ($\Longleftrightarrow$), and "incomparability" ($\not\Longleftrightarrow$) as follows. Attack $A >$ attack B, if attack $A \Longrightarrow$ attack B, and attack B $\not\Longrightarrow$ attack A. Attack $A \Longleftrightarrow$ attack B if attack $A \Longrightarrow$ attack B and attack $B \Longrightarrow$

attack A. Equivalence captures, for instance, cases when the same ("copycat") attack is launched against different instances of the same wimp in different systems. Attack equivalance differs from attack "isomorphism"($\sim$) where we say that attack $A \sim$ attack B if they have the same goals, capabilities and strategies, except that they refer to different types of wimps. Attack A $\Leftrightarrow\!\!\!\!\!\!/$ attack B if attack A $\Rightarrow\!\!\!\!\!\!/$ attack B and attack B $\Rightarrow\!\!\!\!\!\!/$ attack A. Incomparability captures many attack differences including attacks whose goals differ, others whose capability sets differ, and finally those whose strategies differ.

The dominance and incomparability relations naturally lead to partial orders on adversary attacks and hence to *basic* security metrics. It follows that without accurate and complete adversary definitions one cannot define accurate and complete security properties, and without such properties one cannot obtain basic security metrics. Incomplete adversary definitions (e.g., traditional "attack trees" and "kill chains") would not do. It also follows that precise security metrics require wimp definitions and separation from giants, since giants are part of the adversary. Of course, other relations among attacks exist and can be used to define a much richer set of metrics. An orthogonal set of basic metrics arises from (partial) orders among the different types of security-property assurance and assurance evidence.

Using accurate and complete definitions, one can then use traditional proof techniques to perform different types of attack reductions and compositions for different types of wimps. For example, one can formally verify that a wimp has security property $\overline{A}$ in the presence of adversary attack A launched by a giant, as follows. First one verifies $\overline{A}$ assuming that the wimp is isolated from the giant. Then one verifies that the micro-hypervisor (i.e., a basic wimp) supports application wimp isolation [17] and cannot be bypassed by an attack B launched by the giant; i.e., the micro-hypervisor has security property $\overline{B}$. Finally, one proves that if the micro-hypervisor has property $\overline{B}$, then the wimp has property $\overline{A}$ when compiled and registered with the micro-hypervisor and invoked using it. In short, one is able to provide compositional, composability, and additivity proofs, in Rushby's verification terminology for separation kernels [20].

4 Wimps' Dance with Giants

There are many examples of wimp interfaces where it is possible to define all attack strategies for simple goals and small sets of capabilities. Such adversary definitions are not intended scale to the size of commodity systems, compose across networks of services, nor retain their usefulness when new applications are installed. For commodity systems, only incomplete definitions (e.g., traditional attack trees and kill chains) derived from hacking exercises (e.g., red teaming, penetration testing) have been practical to date. Reactive countermeasures to individual attacks, or piecemeal security, is all we could deliver for commodity systems and networks in the past.

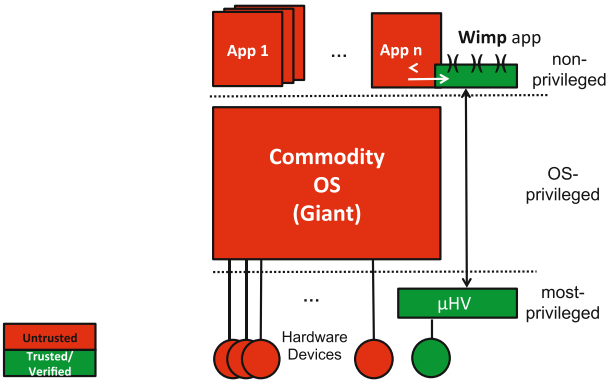

Fig. 3. A wimp-giant isolation architecture based on a micro-hypervisor

How can we do better in the future? A wimp-giant isolation architecture provided by a micro-hypervisor, which operates at a higher privilege level than the giant and hence is isolated from it, is illustrated in Fig. 3. Will wimp-giant isolation be sufficient for accurate and correct adversary definitions, demonstrable security properties, and sound security metrics in practice?

The answer to this questions is decidedly negative. Wimps must compose (i.e. "dance") securely with giants for at least two reasons. First, secure wimps must use services provided by adversary-controlled giants and share platform resources (e.g., I/O, physical memory) with them. This can happen only after wimps efficiently verify the results of those services [29], and the initialization of platform resources in a secure (e.g., malware-free) state [26]. Second, secure wimps could help insecure giants restrict their own (adversary) behavior in specific ways; e.g., prove that certain malicious behaviors are not perpetrated by a giant. For example, wimps have been used to protect cryptographic libraries and key management subsystems against giant misbehavior [17,28], and can also be used to protect application-level reference monitors and cryptographic protocols [6].

The fact that wimp survival depends on collaborating with adversary-controlled giants appears to be paradoxical: wimps can counter all adversary attacks, but only if they use adversary-controlled services from which they have to defend themselves; and to prove that they have not behaved maliciously in certain applications, giants must rely on secure wimps whose operation they attack.

Using Giant Services. To retain all their security properties with reasonable assurance and not become insecure giants, wimps will have to trade very basic system services for small attack surfaces, diminished adversary capabilities, and weak attack strategies. For example, wimps typically lack persistent memory, file system and directory services, network protocols, trusted paths to humans, and isolated I/O services needed to protect applications; see Fig. 4. Placing such

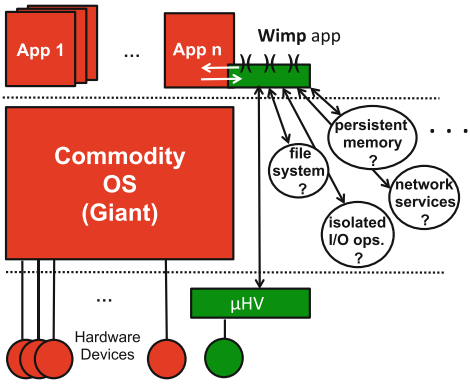

Fig. 4. Examples of missing services from both wimps and micro-hypervisor

services in a trustworthy computing base (e.g., in the micro-hypervisor) to serve wimps would be inadvisable. Trusted computing bases would become bloated, unstable, and devoid of security assurance; e.g., they would often include code of diverse and sometime uncertain origin, such as device drivers.

Note that a different choice of service placement was made in Lampson's *red-green machine* [13]. The red-green machine actually separates *two giants*: an untrustworthy (red) one from a less untrustworthy (green) one. The green giant is a carefully configured, maintained, and connected full-service machine. Although this type of separation can certainly be attained in practice, it does require a trusted-path mechanism to enable careful users to determine the machine they talk to; viz., CMU's Lockdown system [23]. Since the green giants are self-contained, trustworthy red-green communication, and use of the red machine's (efficiently verifiable) services by the green one, is rare. In contrast with wimps, defining and countering all attacks against the green machine remains a daunting and likely unattainable goal.

In principle, one does *not* need to lump services needed by wimps in a green machine. Efficient verification of some system-service results, which enables service implementation in red giants, has been known for over three decades; e.g., cryptographic verification of page integrity [4] and implementation of virtual memory services outside a security kernel in an untrusted operating system. Other efficiently verifiable services, which require only minimal trusted base support, have been proposed more recently; e.g., persistent wimp memory [21], and selected kernel functions for on-demand isolated I/O channels [29]. A wimpy I/O-kernel is illustrated in Fig. 5.

Sharing Platform Resources with Giants. Wimp sharing of commodity platform resources with giants is both useful in practice and fundamentally necessary. It is useful for resources that can be isolated from giants, such as I/O channels and devices, and made available to wimps on-demand. This enables wimps to use only the devices they need and when they need them. Thus, wimps need not

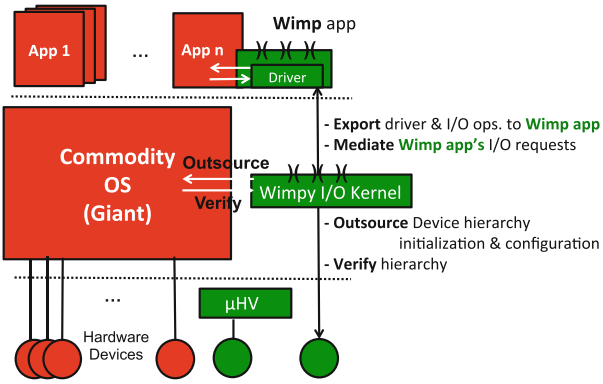

Fig. 5. Wimpy-kernel architecture for on-demand isolated I/O services

include nor rely on unnecessary I/O services, which would dramatically increase their exposure to attacks from giants. However, sharing I/O devices with giants requires wimps to verify the integrity of device firmware and initialize devices to known secure (e.g., malware-free) states after giant use [14,27].

Sharing a hardware platform with the giant is unavoidable for most wimps when dedicated hardware (e.g., co-processors) is unavailable – a common case on commodity platforms. For example, the most basic system wimp, the micro-hypervisor, shares the CPU, memory, and some basic (e.g., DMA) devices with the giant. How can one be sure that giant-inserted malware in memory and device controller firmware does not corrupt the micro-hypervisor *before* the micro-hypervisor boots? To detect and/or prevent this from happening, one needs to introduce the notion of the *verifiable boot*, whereby the micro-hypervisor boots only in a malware-free device state. The notion of verifiable boot is stronger than both *trusted boot* and *secure boot* [18]. Neither trusted nor secure boot provides assurance of malware absence in the *entire device* – not just in directly-addressable processor memory [26] – at boot time and immediately thereafter. Whether the giant infects devices with malware later would become less relevant for a wimp that is able to re-initialize devices that are shared with a giant to a known secure (e.g., malware-free) state on-demand.

It is worth noting that implementing the notion of the verifiable boot on a commodity platform would enable a user to reset a platform to a malware-free state, and a micro-hypervisor and application wimps to execute in an *untampered* execution environment. Furthermore, *on-demand verifiable boot* would enable a user to ensure that application wimps can restart in a malware-free state in the, hopefully unlikely, case of successful penetration by giants. In this case, the giants would be forced to dance "FlipIt" [3] with wimps.

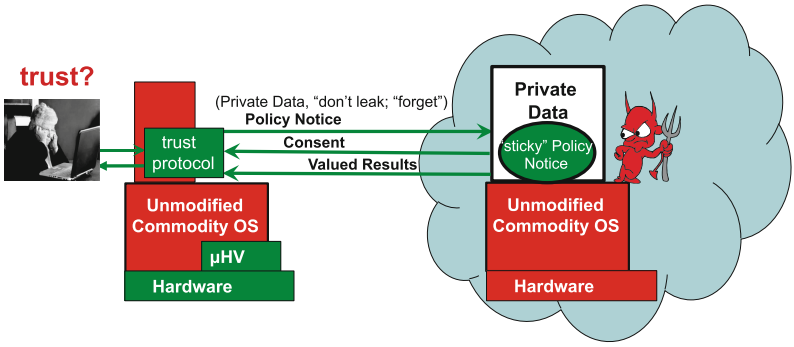

Fig. 6. An interactive trust protocol: sending private data to a giant

5 Wimps and Giants in Networks of Humans and Computers

There is an intriguing but limited similarity between wimp-giant collaboration on commodity platforms and *interactive trust protocols* [7,11] between ordinary users and web services. For example, interactive trust protocols (1) produce value for participants who cooperate; (2) potentially allow malicious participants (i.e., giants) to harm honest participants (i.e., wimps) by employing deception and scams; and yet (3) may have safe states whereby an honest participant can establish beliefs in a malicious participant's temporary trustwothiness, even if traditional security and cryptography techniques cannot be employed. Like in the typical wimp-giant collaboration, interactive trust protocols offer attractive and compelling web services to users. However, dissimilarities are equally evident. Unlike the wimp-giant collaboration on commodity platforms, where a wimp *never* engages services of giants unless it can verify the results of those services, ordinary users can seldom verify the results produced by adversary-controlled services and defend themselves against attacks. Nevertheless, interaction between honest but unsuspecting users and adversaries is desirable, if safe. Such interaction can lead to new trust relations to be formed and potentially can create new value. Safe protocol states can, in principle, provide credible evidence that the adversary-controlled service isn't harming an unsuspecting user. Here, again, the attack definition would benefit from specifying a wimp-giant security game, including the <goals, capabilities, strategies> triple and a model of the adversary's power and privilege.

In the protocol illustrated in Fig. 6, a user delegates her rights to a client-machine wimp, which executes the steps of a trust protocol with an adversary-controlled service; i.e., a giant. The wimp agrees to provide the user's private data, for example personal indentification information, to the giant in exchange for results the user deems to be valuable; e.g., personalized ads, live news, weather and traffic reports. It cryptographically seals a personal privacy policy (i.e., a "sticky") notice onto the private data specifying that they must not be

leaked to third-parties and "forgotten" as specified; i.e., erased from the giant's storage within a certain time/use limit or on-demand. The cryptographic seal and much of the interaction with the web server is done via a wimp on user's machine. This a giant's software would be unable to interfere with the user's actions.

The cryptographic seal has a dual role: first, it prevents the giant from accessing the user's private data unless the giant consents to abide by the user's policy; second, it prevents the user from changing her mind, adding more policies after the giant consents, and then complaining about policy violations. This phase of the protocol represents an undeniable fair exchange, which secures a consent (by the giant) and a policy commitment (by the wimp) thereby assuring mutual accountability. However, mutual accountability does not guarantee giant compliance with the user's privacy policy. The wimp has no way to control the giant's operation and enforce non-leakage and timely data erasure. So why would a user output her user's private data to the giant? Clearly, the user must establish "output trust" in the giant. How could that happen?

First, the user can decrease the risk of giant leakage by *anonymizing* her identity and network address, and ensure that her anonymous identity is *unlinkable* to any other identity she may have used in the past. The user can always change her anonymous and unlinkable identity to reduce the damage caused by giant-saved and leaked private data. Second, the user must ensure that the giant's service is regulated by legal statutes, and thus a non-compliant but accountable giant may be punished. Third, the user could obtain recommendations attesting to the giant's trustworthiness and reputation ratings, which may increase the user's beliefs in giant's trusworthiness in abiding by the user's privacy policy.

Individually, none of the three components of trust establishment between the wimp and the giant offers absolute guarantees of policy compliance by the giant. First, anonymous and unlinkable identities cannot prevent a powerful giant from collecting large amounts of data regarding this user's behavior and linking her identities via *behavioral correlations*. Second, accountability may not necessarily guarantee punishment under the legal statutes and punishment may not necessarily deter a non-compliant giant. However, the user's aversion to betrayal by the giant may be reduced considerably. Third, recommendation systems and reputation ratings can only capture past evidence of trustworthiness but do not necessarily guarantee present or future honest behavior. Nevertheless, research in behavioral economics and practice suggests that all three measures are often sufficient for trust establishment. For this reason, the wimp-giant collaboration suggested by this example may not be as dangerous as anticipated despite the wimp's inability to verify the giant's future actions.

6 Summary

In this paper, we argue that accurate and complete adversary definitions are necessary if the asymmetric advantage of an attacker over a defender is to be eliminated. However, such definitions are likely to be possible only for "wimpy"

software components. We provide a structure for accurate and complete adversary definitions for wimps, which was inspired from similar definitions in cryptography. These definitions yields basic security properties and metrics, and are instrumental in providing security assurance for commodity systems. Although the wimp isolation from giant software components becomes necessary for obtaining such definitions in practice, it is insufficient for wimp survival. To survive in commodity markets, secure wimps must compose with insecure giants. We illustrate a safe way to compose a wimpy I/O kernel with a commodity operating system, and extend the wimp-giant composition metaphor to interactive trust protocols in networks of humans and computers.

Acknowledgments. This paper benefitted from discussions and joint work with Min Suk Kang, Miao Yu, Jun Zhao, and Zongwei Zhou. Their insights are gratefully acknowledged. This work was supported in part by the National Science Foundation (NSF) under grant CCF-0424422 and a gift from Intel Corporation at CyLab. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

References

1. Amoroso, E.G.: Fundamentals of Computer Security Technology, pp, 15–29. Prentice-Hall (1994) ISBN0131089293
2. Bishop, M., Dilger, M.: Checking for race conditions in file accesses. Comput. Syst. **9**(2), 131–152 (1996)
3. van Dijk, M., Juels, A., Oprea, A., Rivest, R.L.: FlipIt: the game of "Stealthy Takeover." J. Cryptology **26**(4), 655–713 (2013). (also in IACR Cryptology ePrint Archive, Report 2012/103, 2012)
4. Gligor, V.D., Lindsay, B.G.: Object migration and authentication. IEEE Trans. Softw. Eng. **SE–5**(6), 607–611 (1979)
5. Gligor, V.D.: On the evolution of adversary models in security protocols (or Know Your Friend and Foe Alike). In: Christianson, B., Crispo, B., Malcolm, J.A., Roe, M. (eds.) Security Protocols 2005. LNCS, vol. 4631, pp. 276–283. Springer, Heidelberg (2007)
6. Gligor, V.D.: Security limitations of virtualization and how to overcome them. In: Proceedings of the 18th International Workshop on Security Protocols (SPW-18). LNCS, Cambridge University, UK, vol. 7061. Springer, March 2010
7. Gligor, V., Wing, J.M.: Towards a theory of trust in networks of humans and computers (transcript of discussion). In: Christianson, B., Crispo, B., Malcolm, J., Stajano, F. (eds.) Security Protocols 2011. LNCS, vol. 7114, pp. 223–242. Springer, Heidelberg (2011)
8. Gupta, S., Gligor, V.D.: Experience with a penetration analysis method and tool. In: Proceedings of the 1992 National Computer Security Conference, Baltimore, Maryland, pp. 165–183 (1992)
9. Howard, M., Pincus, J., Wing, J.M.: Measuring relative attack surfaces. In: Lee, D.T., Shieh, S.P., Tygar, J.D. (eds.) Computer Security in the 21st Century, chap. 8, pp. 109–137. Springer, New York (2005)

10. Hutchins, E.M., Clopper, M.J., Amin, R.M.: Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion Kill Chains. In: Proceedings of the 6th Annual International Conference on Information Warfare and Security, Washington, DC (2011)
11. Kim, T.H.-J., Gligor, V., Perrig, A.: Street-level trust semantics for attribute authentication (transcript of discussion). In: Christianson, B., Malcolm, J., Stajano, F., Anderson, J. (eds.) Security Protocols 2012. LNCS, vol. 7622, pp. 96–115. Springer, Heidelberg (2012)
12. Lampson, B.W.: Software components: Only the giants survive. In: Computer Systems: Theory, Technology, and Applications, pp. 137–145. Springer, New York (2004)
13. Lampson, B.W.: Usable security: how to get it. Commun. ACM **52**, 25–27 (2009)
14. Li, Y., McCune, J., Perrig, A.: VIPER: verifying the integrity of peripherals firmware. In: Proceedings of the ACM Conference on Computer and Communications Security (2011)
15. Manadhata, P.K., Karabulut, Y., Wing, J.M.: Report: measuring the attack surfaces of enterprise software. In: Massacci, F., Redwine Jr., S.T., Zannone, N. (eds.) ESSoS 2009. LNCS, vol. 5429, pp. 91–100. Springer, Heidelberg (2009)
16. Mauw, S., Oostdijk, M.: Foundations of attack trees. In: Won, D.H., Kim, S. (eds.) ICISC 2005. LNCS, vol. 3935, pp. 186–198. Springer, Heidelberg (2006)
17. McCune, J., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., Perrig, A.: TrustVisor: efficient TCB reduction and attestation. In: CMU-CyLab-09-003, March, 2009. (also in Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, May 2010)
18. Parno, B., McCune, J.M., Perrig, A.: Bootstrapping trust in commodity computers. In: Proceedings of the IEEE Symposium on Security and Privacy, May 2010
19. Rogaway, P.: On the role definitions in and beyond cryptography. In: Maher, M.J. (ed.) ASIAN 2004. LNCS, vol. 3321, pp. 13–32. Springer, Heidelberg (2004)
20. Rushby, J.M.: Separation and Integration in MILS (The MILS Constitution). Technical report, SRI-CSL-TR-08-XX, Feb 2008
21. Parno, B., Lorch, J., Douceur, J., Mickens, J., McCune, J.: Memoir: practical state continuity for protected modules. In: Proceedings of the IEEE Symposium on Security and Privacy (2011)
22. Vasudevan, A., Chaki, S., Jia, L., McCune, L.J., Newsome, J., Datta, A.: Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. In: Proceedings of the IEEE Symposium on Security and Privacy (2013)
23. Vasudevan, A., Parno, B., Qu, N., Gligor, V., Perrig, A.: Lockdown: a safe and practical environment for security applications. In: CMU-CyLab-09-011, 14 July 2009. (Also in Proceedings of TRUST, Vienna, Austria, 2012)
24. Schneier, B.: Attack trees. Dr. Dobb's J. **24**(12), 21–29 (1999)
25. Weiss, J.D.: A system security engineering process. In: Proceedings of the 14th National Computer Security Conference, Baltimore, Maryland (1991)
26. Zhao, J., Gligor, V., Perrig, A., Newsome, J.: ReDABLS: revisiting device attestation with bounded leakage of secrets. In: Christianson, B., Malcolm, J., Stajano, F., Anderson, J., Bonneau, J. (eds.) Security Protocols 2013. LNCS, vol. 8263, pp. 94–114. Springer, Heidelberg (2013)
27. Zhou, Z., Gligor, V., Newsome, J., McCune, J.: Building verifiable trusted path on commodity x86 computers. In: Proceedings of the IEEE Symposium on Security and Privacy (2012)

28. Zhou, Z., Han, J., Lin, Y.-H., Perrig, A., Gligor, V.: KISS: "key it simple and secure" corporate key management. In: Huth, M., Asokan, N., Čapkun, S., Flechais, I., Coles-Kemp, L. (eds.) TRUST 2013. LNCS, vol. 7904, pp. 1–18. Springer, Heidelberg (2013)
29. Zhou, Z., Miao, Y.: Dancing with giants: wimpy kernels for on-demand isolated I/O on commodity platforms. In: Proceedings of IEEE Symposium on Security and Privacy, Oakland, CA (2014)