New Directions for Hardware-assisted Trusted Computing Policies (Position Paper)

Sergey Bratus · Michael E. Locasto · Ashwin Ramaswamy · Sean W. Smith

Dartmouth College
Hanover, New Hampshire

July 21, 2008

Abstract

The basic technological building blocks of the TCG architecture seem to be stabilizing. As a result, we believe that the focus of the Trusted Computing (TC) discipline must naturally shift from the design and implementation of the hardware root of trust (and the subsequent trust chain) to the *higher-level application policies*. Such policies must build on these primitives to express new sets of security goals. We highlight the relationship between enforcing these types of policies and *debugging*, since both activities establish the link between *expected* and *actual* application behavior. We argue that this new class of policies better fits developers' mental models of expected application behaviors, and we suggest a hardware design direction for enabling the efficient interpretation of such policies.

1 Introduction

There exists an important design specification and engineering gap to fill before researchers and practitioners can flesh out and experiment with actual policies. Arguably, this gap represents the greatest current challenge

facing TC as a discipline. The gap exists between the hardware elements of the TCG architecture and meaningful policy specifications (specifically, the type and structure of the event stream that such policies operate on). The engineering uncertainly created by this gap has served, in our opinion, as one of the factors that has stymied the development and acceptance[1] of TC platforms. The gap results from an under-specification; the presence of this uncertainty discourages development of higher layer solutions precisely because such solutions have nothing on which to rely.

We claim that the essence of this gap is expressed in the lack of a common specification for both the system of events that TC-based policies need to monitor as well as how the event handlers for such policy systems would cooperate with the basic TCG architecture elements (which are, by definition, passive, and should remain so[2]). To close or reduce this gap, we believe it might be advisable to revisit and enhance the parts of the TC hardware specification that deal with the post-boot life of TC applications.

The ultimate goal of Trusted Computing is to produce development platforms and environments for more trustworthy software. Notably, Trusted Computing promises to offer new kinds of security primitives and trust policies. In contrast, other existing protection initiatives aim to harden software against known classes of exploits and thus to restore trust into existing development models and security primitives[3]. Consequently, an implicit requirement for TC's success is that the policies we discuss must be flexible and easy to write, adapt, and maintain. Indeed, the transition from the TCG architecture fundamentals to high-level policy primitives usable by programmers and administrators proves to be the core significant challenge we highlight.

This challenge has not gone unaddressed. For example, a substantial amount of effort has gone into making TC-based architectures more flexible and expressive, through use of virtualization, on-demand creation of trusted compartments, etc. [BCG+06, HCF04, SPvD05, SZJvD04]. The gist of this

[1]We refer to developer acceptance of TCG as a useful engineering platform rather than acceptance of TCG as a technology by the broader public. In the latter case, concerns over draconian DRM schemes and Big Brother present obstacles beyond the *technical* ones we address in this paper.

[2]For the discussion of this requirement see [Pro05].

[3]For example, address space randomization and stack integrity protections do not prevent developers from accidentally creating buffer and heap overrun conditions. Instead, they just make these vulnerabilities much lower exploitation risks. Thus, the applications' own security checks (rendered useless by these classes of exploits) once again become relevant.

research is to transform the rather inflexible (but the only known effective) way of measuring software by hash digest into more flexible security policy primitives and frameworks. However, we recognize the fundamental problem as a much broader one, to which virtualization and compartmentalization are only partial solutions.

We point to [AAH+07] as an excellent example illustrating the problem of bridging the gap between low-level security primitives as provided by hardware and the OS and the desired application-level security properties such as *decision continuity* and *attribute mutability*. In this example, taken from the healthcare domain, Agreiter et al. highlight the importance of enforcing an access control policy throughout the lifetime of an application process and the ability to change its access privileges based on its observed behavior to date. Their policy implementation mechanism leverages the SELinux MAC to restrict privileged information access to the single process that interprets a dynamic model-based policy and grants or denies access to all other processes, acting, in effect, as a userland "reference monitor", and holding all the logic and state information needed to enforce a dynamic policy. We note the two general design aspects that will be highlighted in our later discussion: (1) reliance on an OS mediation mechanism[4] for the applications' trustworthiness-related access operations and (2) the placement of the policy logic and the corresponding applications state data.

Another example of dynamic runtime enforcement can be found in [BS05]. Again, the policy mechanism's design is driven by the analysis of system events that have the potential of changing the system's trustworthiness.

In this position paper we consider the problem of engineering security primitives and enforcement mechanisms from several unusual angles, and attempt to distill the qualities that are shared by primitives and mechanisms that proved successful and attracted a substantial following among developers.

In particular, we relate the well-known definition of trust in a computing system as *relying on the system to behave as expected* to the common and familiar developer experience of debugging software to link its expected behavior to its actual behavior.

We argue that developers' knowledge of their program's expected behaviors, and, more importantly, of behaviors that they trust to *never* occur

[4]The Linux Security Modules (LSM) system that underlies SELinux's syscall hooks.

while the system remains trustworthy, is a great and mostly untapped source of meaningful application policies. We believe that allowing developers to express this knowledge in policies will help close the above mentioned engineering gap.

We suggest that in order to tap into this resource, future additions to the TCG architecture should provide developers with ways to express policy conditions similar to those used by advanced debugging tools, such as *DTrace* and its extensions. These conditions, of course, will need to be efficiently monitored and enforced without ruining performance; we speculate on how an appropriate enforcement mechanism can be achieved with some extra hardware's help.

2 Policy Engineering

This issue of *policy engineering* presents a clear and non-trivial challenge to systems designers who aim to produce strong and usable security primitives for the use of application developers.

The discipline of software engineering has developed arguments explaining why some design and programming practices endure better than others. More importantly, software engineering has developed practices and tools to make it easier for programmers to produce, debug, analyze, and modify software aimed at non-trivial objectives.

By contrast, there is yet hardly any comparable analysis to explain why some proposed security primitives have much better adoption records than others, let alone tools capable of handling comparably complex behaviors. We attempted to present an instance of such analysis comparing SELinux and virtualization-based policy solutions [BFMS07]. We apply similar analysis [BDSS08] to address the TOCTOU problem in the TCG architecture.

We note that the challenge faced by proponents of novel software engineering primitives is the same as that facing designers of new security primitives: both require adoption by a wider community of developers who must find the proposed semantics natural and only a minor burden compared to developers' current practices.

In particular, we make the following historical observations with regard to security policy mechanisms. We regard them as fundamental to policy engineering, in much the same way as desirability of code reuse, encapsulation, polymorphism, *etc.*, are fundamental observations that lead to maturation

of the object-oriented programming paradigm in software engineering.

1. **Policy design is event-centric.** Defining a manageable set of events for the policy mechanism to monitor and control is crucial to engineering usable and effective policies, because it ultimately determines which security goals can and cannot be easily expressed in the policies.

2. **Context precision is critical to the processing of event streams.** The common trait of successful policy mechanisms is to limit the amount of information that needs to be processed at its event-based decision points to just that relevant to the security goal, and no other. Having to deal with too many pieces of information weighs heavily on developers, because, instead of additional flexibility, it likely translates to having to classify all the combinations of their values as either conforming to or contravening the policy goals, which is serves as a deterrent to adopting the policy mechanism in question.[5]

3. **A little semantic annotation goes a long way when supported by OS and hardware.** Significant practical advances in improving trustworthiness of software followed from a combination of a new lower-level security primitive (*e.g.*, a new OS kernel system call or hardware trapping capability) and a small additional amount of code annotation by programmers. This annotation expressed some semantic security-related properties of resulting binary code or data objects (*e.g.*, "after this point the process no longer needs these elevated privileges", or "this data is not supposed to change within the lifetime of this process" or even – on the ABI level – "this area of memory does not contain any executable code"), and was automatically translated to binary code or data representation by the compiler toolchain.

4. **Strength through cross-layer amalgamation.** The actual machine execution of the program's logical flow becomes a combination of hard-coded fast immutable logic performing the bulk of the necessary event-handling tasks and of program code modified to seamlessly integrate with it. For example, with the introduction of virtual address translation this flow includes a sizeable component of logic performed inside

[5]SELinux's strict policies that require the system administrator to classify an ever-increasing amount of program's file accesses are an example of such quandary.

the MMU. The use of `x86` segmentation in the Linux kernel strengthening patches such as *OpenWall* and *PaX* provides another example. A novel, effective event system usually introduces an extra computational load on the system; however, most of it can be offset by conceptually simple hardware changes of manageable complexity.

We note that the TCG specification concentrates primarily on load-time static software measurements as a means of ensuring the trustworthiness of a process. Consequently, it needs to be complemented by a mechanism that intercepts and mediates such transitions in the running program's state that can render it untrustworthy.

We call such transitions *trust events* and note that, rather than being arbitrary asynchronous OS-level events or system calls, they should be defined for each application as changes in its state that the developer "trusts will not happen" when designing the logic that protects the applications most valuable and sensitive information, its "crown jewels". Such assumptions about the application's and environment's behavior may include pure userland events not mediated by the syscall mechanism, such as writes to or even reads from certain data objects.

It should also be noted that not all OS-level events necessarily have the same impact on the application's trustworthiness, and thus mediating them is not equally important for ensuring it. In other words, the concept of *sensitive and trusted* data for an application need not necessarily coincide with *being accessed through the OS kernel.* SELinux implicitly assumes the latter, and this implicit assumption (and the resulting need to describe all allowed accesses by an application in order to get any degree of protection), in our opinion, leads to severe usability issues.

Once the developer has formulated what constitutes trust events for his application, these events must be monitored and mediated cheaply and "inline". This brings us to the next crucial component of a policy mechanism: system traps.

3 Traps and Security

We next highlight the relationship between event trap semantics, the implementation of a trap system, and policy goal formulation and enforcement. At first glance, the connection between a trap system for a particular platform

and the security properties of that platform may not seem obvious. They are, however, directly and intimately related.

We believe that it is natural to formulate security properties as those preserved across normal transitions in the system's state space, given that the system starts in a trustworthy state. Abnormal transitions should cause traps, after which the system's state may no longer be considered trustworthy or "secure." Accordingly, trap handlers contain much of a security system's functionality.

For security policies, events that correspond to the system's transitions between trusted states play a similar central role in the design and implementation of the policy mechanism. Namely, the policy mechanism is charged with allowing only "safe" transitions that preserve the desired security properties. While such mechanisms can be implemented purely in software, in practice they rely on hardware-supported traps whenever possible, to let application code execute at full speed between mediated events, and as to provide additional assurance of separation between the more and less trusted parts of the system. In practice, therefore, traps form a core mechanism upon which to implement security policy interpreters. As such, they directly or indirectly affect all aspects of the latter. The details of the trap system shapes, *de facto*, the capabilities and performance of the policy system.

3.1 Traps and Debugging

Informally speaking, the process of debugging an application has much in common with the process of enforcing a policy. Instead of "trustworthiness", a bug-hunter tries to ensure that the system behaves according to her mental model of what the code is supposed to do and catch the moment when it begins to deviate[6] from that model. That moment — more precisely, that event — is assumed to be the manifestation of the hunted bug.

Simply put, debugging is the activity that establishes the link between the **expected** *application behavior and its* **actual** *behavior. But so is the enforcement of a security policy!*

[6]We note that behaving as expected is one of the definitions of "trust." Thus a "bug" in the program, from the programmer's point of view, is exactly what breaks the trust.

We believe that this connection has deep implications for future policy design. In our experience, many developers, despite having a reasonably good idea of what constitutes the "crown jewels" and the "worst nightmare" of their applications, and indeed expressing it in various ways throughout the debugging and testing process, cannot easily impart such knowledge to runtime environments. Yet, this knowledge would be eminently useful for a security policy intended to preserve the process' trustworthiness, even as an "80/20 percent solution" that counters only the more frequent threats.

Of course, the programmer's mental model of a program's intended behavior can be more complicated than that of its security properties. As a result, the set of events that the developer needs to monitor to debug the program can be harder to describe than the set of security-related events that require mediation. As a result, debugging likely needs much more flexible support than security policy enforcement. Yet, the available hardware support on commodity platforms remains in its infancy (to put it somewhat bluntly, in an equivalent of the Stone Age).

We believe that the continued lack of a flexible means to describe events relevant to debugging is caused by the absence of a more comprehensive set of hardware primitives. Although software systems can provide this rich set of primitives (Pin [LCM+05] is a good example), real production software constantly pushes the limits of computing speed, making the use of software–level debugging unattractive for such systems. Therefore, we believe that a flexible hardware trap system (one that allows execution to proceed at the highest possible CPU speeds until an event of interest occurs) is a necessary condition of increasing trustworthiness — and, therefore, security, of software.

3.2 Tracing and policy

We point to an interesting use of the OS's tracing functionality, meant primarily for debugging, to monitor security-related properties of an application process. Beauchamp et al. demonstrated[7] *RE:Trace* and *RE:Dbg*, extensions of *DTrace*, that allowed the user to express complex application logic conditions involving both user-level and kernel objects. Among other trustworthiness-related conditions, they were able to catch exploitation of

[7] See [BW08], also `http://blog.poppopret.org/?m=200806` for the updated version of their results and presentation.

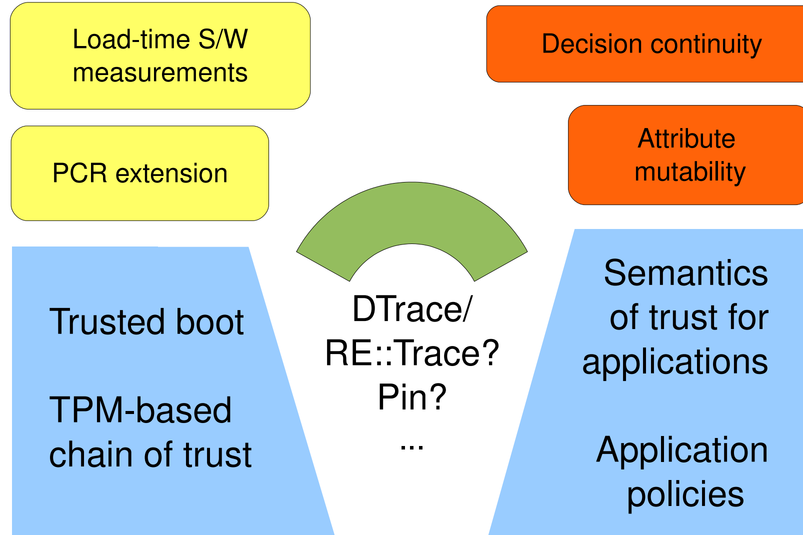

Figure 1: *Bridging the Trust Policy Engineering Gap.* Since the requirements for dynamic, application-level trust assessments outstrip the available capabilities of standard TCG hardware, we propose debugging-like primitives to help reason about trust and store related state.

vulnerabilities via stack and head overflows and suspend the compromised processes.

We note that DTrace itself can be a powerful tool for expressing *auditing* policies that enforce application logic-level conditions. In particular, a developer can use it to specify conditions that are trusted to never happen in the application's trustworthy state, and register the loss of trustworthiness should the respective probe "fire".

Of course, due to the fundamental architectural properties of DTrace, this approach would work only for auditing, since the probes are processed asynchronously, and cannot be used to mediate the respective trust events that triggered them (moreover, there are no specific guarantees as to how soon after the event a specific probe would fire). These properties of the DTrace architecture are quite deliberate and are due to performance considerations – they recognize the fact that full debugger-style mediation of a process, even

in the OS kernel, cannot currently be fast enough to be compatible with acceptable performance expectations.

Still, we recognize the great potential of specifying policies on such higher level, which is also a much better match for developers' mental models of their applications' expected (and explicitly *not* expected) behaviors. Thus we propose to turn to hardware for help in enforcing such policies.

4 Proposed Hardware Features

Changing the way systems trap and service memory events requires both programmability and speed. In essence, we need an architecture that simultaneously allows more complex analysis and a faster overall (amortized) trap service speed. We propose an architecture that contains two primary components. First, an FPGA configured to act as a memory event stream parser interacts with the CPU and MMU to obtain a stream of memory events and a series of interrupts. Second, a memory event analysis policy is loaded into the memory of the FPGA to direct the actions of the FPGA. With the architecture in Figure 2, we hope to satisfy the twin demands of flexible analysis and better trap performance. We point to successful uses of FPGA implementations of application-aware policies to improve trustworthiness of special purpose applications [IKP+07].

The capabilities of FPGA logic enables the TC community to define a richer set of events and their contexts — contexts that previously could only be defined and handled by debuggers (*e.g.,* watchpoints that "fire" only under particular circumstances or that depend on the state of the process context). While policy designers could express many security goals quite naturally as conditions for a tracing debugger to check, the overhead of doing so makes efficient policy enforcement entirely infeasible. The introduction of the FPGA, however, makes it possible to trace a limited set of such conditions efficiently, since the FPGA provides both the place to store necessary state information and fast logic to update and check such information.

5 Self-healing Perspective

We believe that one of the important considerations in designing the TCG-compatible event systems and policy mechanisms should be leaving room

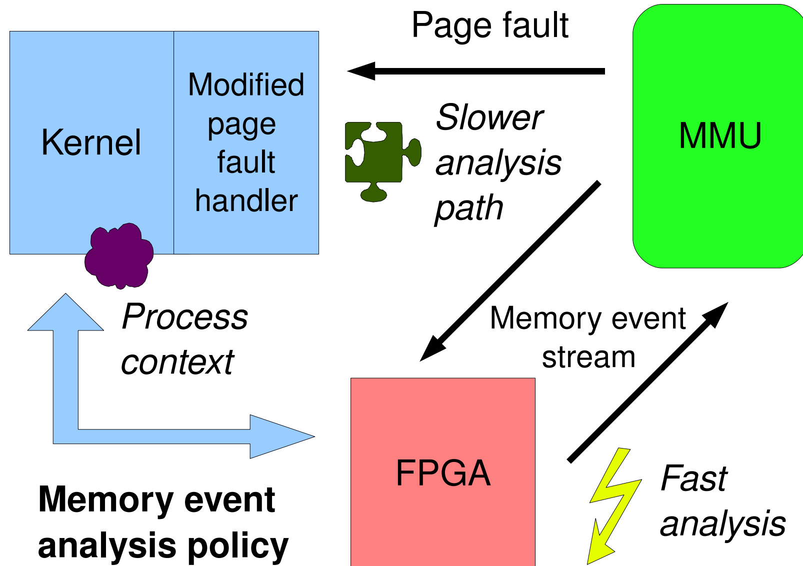

Figure 2: *An Architecture for Efficient Trust Policy Enforcement.* We propose an architecture aimed at servicing most basic policy events at machine speed (rather than serviced by a software interrupt) by trapping and interpreting policy events within an FPGA.

for *self-healing*. In production environments, security goals tend to compete with availability requirements in that security mechanisms can represent a negative impact on performance. Availability, however, is itself a cornerstone security requirement. In our opinion, it would be disadvantageous for TC to settle on policy mechanisms that exclude self-healing as a way of providing availability.

6 Conclusion

We believe that the current capabilities of TCG infrastructure exhibits a gap between the needs of security policy writers and the existing TC hardware specification. We argue that specifying a system of events, trapped and monitored by a mechanism compatible with the fundamental passive elements of the TCG architecture is necessary for development of flexible and usable

Trusted Computing policies.

Further, we point to the basic similarities between debugging, trust and policy enforcement. We argue that allowing developers to express their applications' trustworthiness assumptions in forms such as might be used for debugging with DTrace or Pin could become a useful source of dynamic policies, connecting expected behavior of a running applications and its trustworthiness. We note that the role and knowledge of a developer in achieving the latter is crucial, but to date has received little attention in the design of policy mechanisms.

Finally, we propose the use of FPGAs as a basic primitive to efficiently handle memory and process-related trust events that would play a central role in the proposed class of policies.

7 Acknowledgements

This work was supported in part by the National Science Foundation, under grant CNS-0524695, the U.S. Department of Homeland Security under Grant Award Number 2006-CS-001-000001, and the Institute for Security Technology Studies, under Grant number 2005-DD-BX-1091 awarded by the Bureau of Justice Assistance. The views and conclusions do not necessarily represent those of the sponsors.

References

[AAH+07] B. Agreiter, M. Alam, M. Hafner, J.-P. Seifert, and X. Zhang. Model Driven Configuration of Secure Operating Systems for Mobile Applications in Healthcare. In *In Proceedings of the 1st International Workshop on Model-Based Trustworthy Health Information Systems*, 2007.

[BCG+06] Stefan Berger, Ramon Caceres, Kenneth Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vTPM – Virtualizing the Trusted Platform Module. In *15th Usenix Security Symposium*, pages 305–320, 2006.

[BDSS08] Sergey Bratus, Nihal D'Cunha, Evan Sparks, and Sean Smith. TOCTOU, Traps, and Trusted Computing. In *Proceedings of the TRUST 2008 Conference*, March 2008. Villach, Austria.

[BFMS07] Sergey Bratus, Alex Ferguson, Doug McIlroy, and Sean Smith. Pastures: Towards Usable Security Policy Engineering. In *ARES '07: Proceedings of the The Second International Conference on Availability, Reliability and Security*, pages 1052–1059, Washington, DC, USA, 2007. IEEE Computer Society.

[BS05] Kwang-Hyun Baek and Sean W. Smith. Preventing theft of quality of service on open platforms. Technical Report TR2005-539, Dartmouth College, Computer Science, Hanover, NH, May 2005.

[BW08] Tiller Beauchamp and David Weston. DTrace: The Reverse Engineer's Unexpected Swiss Army Knife. Blackhat Europe, 2008.

[HCF04] V. Haldar, D. Chandra, and M. Franz. Semantic Remote Attestation: A Virtual Machine Directed Approach to Trusted Computing. In *USENIX Virtual Machine Research and Technology Symposium*, 2004.

[IKP+07] Ravishankar K. Iyer, Zbigniew Kalbarczyk, Karthik Pattabiraman, William Healey, Wen-Mei W. Hwu, Peter Klemperer, and Reza Farivar. Toward Application-Aware Security and Reliability. *IEEE Security and Privacy*, 5(1):57–62, 2007.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of Programming Language Design and Implementation (PLDI)*, June 2005.

[Pro05] G.J. Proudler. Concepts of Trusted Computing. In Chris Mitchell, editor, *Trusted Computing*, pages 11–27. IET, 2005.

[SPvD05] Elaine Shi, Adrian Perrig, and Leendert van Doorn. BIND: A Fine-Grained Attestation Service for Secure Distributed Systems. In *IEEE Symposium on Security and Privacy*, pages 154–168, 2005.

[SZJvD04] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *USENIX Security Symposium*, pages 223–238, 2004.