CARMA: A Hardware Tamper-Resistant Isolated Execution Environment on Commodity x86 Platforms

Amit Vasudevan Jonathan McCune
James Newsome Adrian Perrig
CyLab / Carnegie Mellon University
{amitvasudevan, jonmccune, jnewsome, perrig}@cmu.edu

Leendert van Doorn
Advanced Micro Devices
Leendert.vanDoorn@amd.com

Abstract

Much effort has been spent to reduce the *software* Trusted Computing Base (TCB) of modern systems. However, the *hardware* TCB remains complex and untrustworthy. Components such as memory, peripherals, and system buses may become malicious via firmware compromise, a malicious manufacturer, a malicious supply chain, or local physical tampering. We seek to reduce the hardware TCB to a minimal set of hardware components that must be trusted. We describe the design and implementation of an isolated execution environment on commodity x86 platforms that only relies on the CPU, without needing to trust the memory, buses, peripherals, or any other system components.

Categories and Subject Descriptors D.4.6 [*Software*]: Operating Systems-Security and Protection

General Terms Security, Systems

Keywords TCB reduction, Secure Execution, On-die Execution, Cache-as-RAM

1. Introduction

Much effort has been spent to reduce the *software* Trusted Computing Base (TCB) of modern systems. However, there remains a large and complex *hardware* TCB, including memory, peripherals, and system buses. There are many realistic adversary models where this hardware may be malicious or compromised. Thus, there is a practical need to determine whether we can achieve secure program execution in the presence of not only malicious software, but also malicious *hardware*.

ASIACCS '12, May 2–4, 2012, Seoul, Korea.

Attack Level	Type	Description
AL0	Remote	Malware in application
AL1	Remote	Malware in Operating System
AL2	Remote	Malware in devices' firmware
AL3	Local	Physical attachment of malicious peripheral equipment (e.g., FireWire, Thunderbolt, PC Card, or PCIe card)
AL4	Local	Low-speed bus (e.g., LPC bus, 33 MHz) tampering, e.g., interpositioning or TPM reset attacks [8, 20]
AL5	Local	High-speed bus (e.g., memory or PCI bus, 1 GHz or more) tampering, e.g., eavesdropping to attack data secrecy, injection to attack code and data integrity
AL6	Local	Malicious CPU

	AL0	AL1	AL2	AL3	AL4	AL5	AL6
Operating system primitives							
Information flow controlled OS	Y	Y					
Verified Kernel	Y	Y					
Programming language primitives							
Software Fault Isolation (SFI)	Y	Y					
Sandboxes (NACL)	Y	Y					
Hardware primitives							
DRTM (Flicker [13], TV [12])	Y	Y	Y	Y			
TRESOR [14]	Y	Y					
Specialized processor (Mondriaan [25], Aegis [22], Cerium [1])	Y	Y	Y	Y	Y	Y	
CARMA (this paper)	Y	Y	Y	Y	Y	Y	
Cryptographic primitives							
Secure Multi-Party Computation [26]	Y	Y	Y	Y	Y	Y	Y

Figure 1. Adversary attack levels on commodity x86 platforms and comparison of existing approaches for defending against them

Specifically, we consider the adversary attack levels as shown in Figure 1. These attack levels correspond to an increasing level of attack cost and complexity.

The current frontier in remote attacks (AL2) is to attack peripheral devices. Arrigo et al. demonstrate how a compromised network card can use the GPU via peer-to-peer PCI communication, creating a botnet node with powerful

computation and communication abilities, completely out of reach of the OS [23].

For local attacks, simply booting from removable media or plugging in a malicious device that obtains DMA access to all of memory (AL3) is feasible without tools and with a trivial amount of training. More sophisticated attacks require opening the computer system and directly attaching to a bus, either by resetting the bus or by interposing on the bus signals (AL4). The much faster speed of the memory and PCI buses requires more sophistication in terms of equipment and attacker skill (AL5). Supply-chain infiltration may be a more realistic source of such malicious components [15, 24]. While a simple handheld device (e.g., a smartphone) could be programmed to interface with low-speed buses, specialized equipment is needed for the high-speed buses.

Given this framework for increasing attack levels, it is interesting to consider existing approaches for defending against these threats. Figure 1 describes categories of current defense mechanisms, and which attack levels they are effective against.

TRESOR [14] is one recent system intended to defend againt physical cold-boot attacks where memory chips are frozen and extracted from a running system, by storing cryptographic keys in debug registers within the CPU. However, TRESOR assumes code integrity (i.e., that the attacker is unable to modify the contents of memory while the system is still running). We grant the attacker this capability in AL3 and above, and seek to achieve code integrity even against such stronger adversaries.

Secure co-processor hardware achieves strong secrecy and integrity properties against even malicious peripherals and buses (up to AL5), but these solutions represent expensive specialized hardware [1, 22, 25].

Secure multi-party communication can address all of our proposed attack levels, including malicious CPUs (AL6), and does not require any trusted hardware. However, its highly inefficient computation (about six orders of magnitude slowdown) prohibits viable deployment today [3].

Contributions. The challenge that we tackle in this paper is how to defend against a sophisticated adversary with physical access to the host; i.e., AL5. Our goal is to achieve efficient execution, code integrity, data integrity and secrecy against a class AL5 attacker on commodity hardware. Towards this end we have developed CARMA, a secure execution primitive that removes system buses and peripherals from the TCB, requiring us to trust only the CPU and a simple inexpensive external verification device. We implement this basic primitive on an off-the-shelf PC (§3).

2. CARMA Design

On current x86 hardware platforms, the TCB for trustworthy execution primitives includes various low-level system components (CPU, memory-controller, IOMMU, TPM, buses). The goal of CARMA is to achieve code integrity, launch

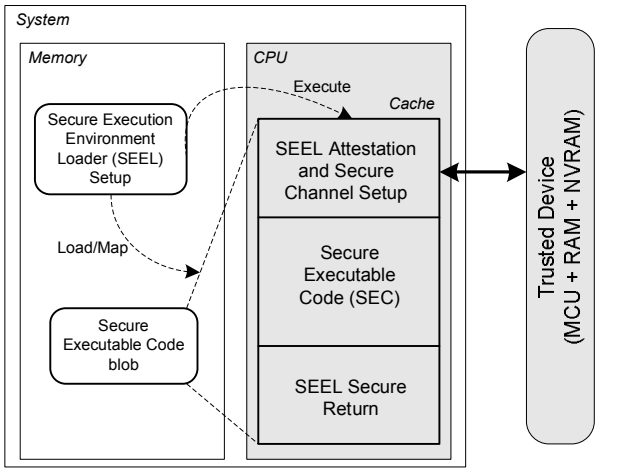

Figure 2. CARMA Architecture: A Secure Execution Environment, running a Secure Executable Code (SEC) module, is created entirely within CPU cache. An external Trusted Device forms the initial root-of-trust to verify environment instantiation, SEC load, and SEC execution. The TCB includes only the CPU and the Trusted Device.

point integrity, and data integrity/secrecy, with a dramatically reduced TCB compared to previous work, so that only the CPU needs to be trusted. Consequently, our desired properties will hold even if all other hardware components are malicious, including the TPM, peripherals, memory, etc. Since the CPU is the primary system component that executes instructions, it is natural to trust it for execution.

Figure 2 shows our high-level architecture. Our central idea is to execute entirely inside the CPU enclosure, leveraging a special execution mode called *Cache-as-RAM*. This allows the CPU to act as a self-contained computing environment, without having to rely on external untrusted RAM. We use a simple external trusted device (TD) to act as a root of trust to bootstrap the environment. We employ a SoftWare-Only Root of Trust (SWORT) mechanism to verify that the intended code is loaded, and to establish a shared secret that we use to create a secure communication channel.

2.1 Background: Cache-as-RAM (CAR)

Commodity CPUs include a cache subsystem to reduce the average time to access memory (DRAM). The cache is a smaller, faster, on-die memory which stores copies of the data from the most frequently used DRAM locations. Most modern desktop and server CPUs have at least three independent caches: an *instruction* cache to speed up executable instruction fetch, a *data* cache to speed up data fetch and store, and a *translation lookaside buffer* (TLB) used to speed up virtual-to-physical address translation for both executable instructions and data. These caches are usually organized as a hierarchy of one or more cache levels (L1, L2 and L3).

During system startup, the BIOS first initializes the CPU, the cache subsystem, the memory controller (DRAM) and then proceeds to initialize other hardware platform elements (PCI, SMBus, peripherals, etc.) before transferring control to the operating system. The BIOS is stored and accessed as read-only memory on current platforms.[1] This presents an interesting problem to the early stages of the BIOS execution (CPU, cache and DRAM initialization) which require a read-write data area and stack.[2] To alleviate this problem, CPUs allow portions of the cache subsystem to be addressed as though they were DRAM until the DRAM controller is initialized in the system. This mechanism is dubbed *Cache-as RAM* (CAR) and is employed by most BIOSes today [11].

2.2 CARMA Approach

Typical CAR usage today is to instantiate a temporary one-shot read/write data area during system bootup. However, we found that by carefully programming the CPU, we can use the cache as a general-purpose memory area, supporting simultaneous code-execution and data read/writes. This provides us with an isolated execution environment entirely within the CPU. Our findings also seem consistent with publicly disclosed high-level information about the dynamic launch facility in current x86 CPUs employing some form of reserved memory inside the CPU for authenticated code modules [5] – indicating that our scheme is general purpose and applicable to commodity x86 CPUs. More details on the actual implementation are available in §3.

We design an execution model where a Secure Execution Environment Loader (SEEL) executes a Secure Executable Code (SEC) module (Figure 2). The SEEL sets up the CAR environment within the CPU and loads the SEC into the CAR region.

The isolated execution environment is not very useful without some means of communicating securely with an external entity. In our design, that entity is a simple external device. We call this device the Trusted Device, or TD. Before passing control to the SEC module, the SEEL first attests to the TD that the SEEL has been loaded correctly, and sets up a secure channel between the SEEL and the TD.

We next describe each step of instantiating, using, and cleaning up a CARMA session.

Secure Environment Setup. The SEEL and SEC module are first loaded into the CPU, and the SEEL begins executing. The SEEL configures the CPU into Cache-as-RAM mode, ensuring that subsequent execution will be entirely within cache and not interact with untrusted RAM.

Attesting to Secure Setup. Since the Secure Environment Setup was necessarily initiated by untrusted code, and the SEEL and SEC were loaded across untrusted buses, we must securely attest the loaded code to the TD. Since the CPU has no on-board hardware root of trust (e.g., the TPM is external, across an untrusted bus), we use a SoftWare-Only Root of Trust [2, 4, 7, 9, 10, 18, 19, 21]. We extend the Pioneer [17] system, which is an implementation of SWORT on x86-class systems. In our setting, the TD is aware of the expected memory content of the cache and sends a challenge to the SEEL. The checksum function in the SEEL includes execution state of the CAR environment in the checksum computation. Any deviation from the expected loaded code or CPU state will give an incorrect checksum result. As with all SWORT checksum functions, the function is designed such that a modified checksum function that attempts to "lie" will take measurably longer to compute. Consequently, a correct checksum received before a threshold time indicates to the TD: the secure CAR environment is correctly set up, SEEL and SEC code integrity, and SEEL execution integrity.

We must ensure that an attacker cannot use a faster processor than the CPU to calculate the correct checksum value. This attack is partially prevented by the SWORT function being designed such that no other local component can compute it as quickly as the CPU. Although a local graphics card can also execute at high speed with high parallelism, the sequential nature of the SWORT checksum function and the optimized implementation for the target CPU precludes such an attack.

We must also assume that the attacker does not use a faster *remote* system to compute the checksum. While the user could validate this assumption by physically disconnecting network access, this is clearly not ideal. A possible way of removing this assumption is to use Li et al.'s strategy employing several short-running checksum computations instead of a single long-running checksum computation, such that the latency to communicate with a remote faster system would be greater than any computational advantage [10].

We must also assume that the attacker does not obtain a linear speedup of the checksum computation sufficient to overcome the performance penalty of forging the checksum. The attacker may attempt this attack by overclocking the processor or by physically replacing it with a faster model. The user can attempt to detect such attacks by physical inspection of the machine. Such attacks can be made less feasible by designing SWORT checksums that have a higher performance penalty for forging attacks. Gardner et al. propose a promising SWORT function designed such that a forgery attack would have many more cache misses than the unmodified function, resulting in a 50% performance penalty [2].

Secure Channel Setup. We adapt the Software-based Attestation for Key Establishment (SAKE) protocol [16] to set up a shared secret key between our secure CAR execution environment and the TD. This shared secret key is used to derive an encryption and authentication key to provide secrecy and integrity for all communication. The main challenge in this context is to prevent a Man-In-The-Middle

[1] Typically EEPROM or Flash memory.

[2] Most BIOSes today are byte-addressable as RAM, so execution can proceed from power-up without the DRAM controller being initialized.

TD :	$a \overset{R}{\leftarrow} \{0,1\}^{\ell}$
	$c = g^a \bmod p$
	$T_1 =$ Current time
$TD \rightarrow CPU$:	$\langle c \rangle$
CPU :	$z =$ checksum over SEEL and SEC
	$b \overset{R}{\leftarrow} \{0,1\}^{\ell}$
	$d = g^b \bmod p$
$CPU \rightarrow TD$:	$\langle d, \text{MAC}_z(d) \rangle$
TD :	$T_2 =$ Current time
	Verify $(T_2 - T_1) \leq$ Time thresh
	verify MAC based on correct z
	$K_{TD,CPU} = d^a \bmod p$
CPU :	$K_{TD,CPU} = c^b \bmod p$

Figure 3. CARMA Secure Channel Setup uses a slightly modified SWORT attestation protocol for key establishment [16]. A private Diffie-Hellman key of length ℓ is chosen at random for each protocol execution.

(MITM) attack, as the CPU and TD cannot authenticate each other. The idea that SAKE proposed is to use the checksum of software-based attestation as a short-lived shared secret to bootstrap an authentic public key. Since SAKE was designed for sensor nodes, we can take advantage of the powerful CPU to simplify the design. Specifically, the protocol in CARMA (Figure 3) performs a Diffie-Hellman key exchange, where the CPU uses the checksum z computed via software-based attestation to compute a MAC of its public key d. Since no other entity can compute z as quickly as the CPU, z represents a short-lived shared secret.

However, numerous challenges still exist that need to be addressed.

(1) Authentication of the CPU by the TD is still required; how can the TD ensure that indeed the result originates from the local CPU? In SAKE, the checksum computation includes a unique local value, such as a processor version number or silicon ID. Since we do not have either reliably available on current x86 CPUs, we need to resort to a different approach. We rely on the assumption already discussed in the secure setup attestation step: that the attacker has no other device available that can compute the checksum as fast as the CPU and with sufficiently low communication latency.

(2) An adversary could steal the private key before attestation starts, as it can inspect the state of the random number generator within the CAR environment, thus predicting the value of b. To thwart this attack, we make use of a hardware random number generator within the CPU such as the Intel Digital Random Number Generator (DRNG) [6]. Without built-in random numbers, it may be possible to leverage unpredictability of performance counters as entropy sources.

(3) An adversary would move the expensive computation of g^b before the checksum computation, thus saving time and achieving faster checksum computation. This attack can be defeated by performing the checksum computation long enough such that the time overhead of the fastest adversarial function would still be above the time threshold.

Secure SEC Execution. Once the secure channel is established, the SEEL receives inputs to SEC from TD, executes SEC (offering launch point integrity), and returns SEC outputs to TD. Since the attestation function offers code integrity and SEEL execution integrity, we obtain the launch point integrity as the SEC starts execution at the correct location with the correct executable. Since the checksum function also verifiably turns off all interrupts and exceptions, the execution cannot be disrupted by malware.

Secure Return. After execution, SEEL erases all Cache-as-RAM execution state and resumes normal execution.

3. Implementation and Evaluation

We have developed a proof-of-concept implementation that demonstrates the feasibility of realizing our CARMA approach (§2.2) on a commodity PC platform. Our current prototype implements the secure execution setup, secure SEC execution and secure Return components of our execution model (§2.2). We note that the remaining execution model components, the secure setup attestation and secure channel setup, have existing stand-alone implementations [16, 17]. Adapting them to our prototype should be relatively straightforward based on our design. The CARMA prototype currently runs on a AMD Family 10h CPU and is implemented as a custom BIOS. We used a Tyan S2912e motherboard and coreboot[3] as our development platform. We now proceed to describe the details of our SEEL and the SEC implementations.

The SEEL runs on the Boot-strap Processor (BSP) and sets up the L2 cache as general purpose memory (for code and data read/writes). It first locks the CPU cache subsystem by preventing write-backs and other out-of-order CPU operations (e.g., branch prediction and speculative loads). The SEEL then sets up CPU memory addressing and caching policies to address the L2 cache and maps the SEC code and data into the cache, and transfers control to the SEC code entry-point. When this happens, the SEC code is executing entirely within the CPU cache along with associated data.

Our Secure Executable Code (SEC) is a simple application that prints a "Hello World!" string through the serial port by employing the legacy `in/out` I/O instructions to communicate with the UART. The SEC contains a SEEL epilogue code, that it transfers control to, once the SEC is done with its processing.

The SEEL epilogue code tears down the CAR environment in a secure fashion and allows normal program execution. The SEEL epilogue code first disables the CPU cache, clears contents of MTRRs and programs the L2 cache-subsystem to the state at reset (i.e., no code caching). CPU cache-invalidation, out-of-order execution primitives

[3] `http://www.coreboot.org/`

(branch-prediction, speculative loads and stores) and self-modifying code logic are then enabled before enabling the CPU cache and resuming normal execution.

We confirmed the instantiation of the cache-as-RAM (CAR) environment by removing the DRAM modules from our prototype system and by setting up a performance counter to keep track of L2 cache evictions while runnning our SEC code. For a successful instantiation of the CAR environment, there must be no L2 cache evictions since the L2 cache is being used as memory. Our tests revealed that the value of the performance counter was always 0 during the SEC execution. This indicates that there were no cache evictions and confirms that the SEC is executing entirely within the CAR environment.

4. Conclusions

The lack of isolation mechanisms on modern commodity systems results in an inherently interconnected system where a single malicious component can control the other components and render the entire system compromised. This raises the question of whether it is possible to reduce the hardware trusted computing base (TCB) on such commodity systems.

We demonstrate with the CARMA design that it is indeed possible on a commodity system to remove from the hardware TCB the memory, memory controller, system buses, and peripherals. This leaves in the hardware TCB only the CPU itself and a simple external device. The resulting secure execution environment has guaranteed code integrity, launch point integrity, and data integrity and secrecy.

CARMA offers an exciting execution platform for secure computation that we plan to explore in our future work.

References

[1] B. Chen and R. Morris. Certifying program execution with secure procesors. In *Proceedings of HotOS*, 2003.

[2] R. W. Gardner, S. Garera, and A. D. Rubin. Detecting code alteration by creating a temporary memory bottleneck. *IEEE Transaction on Information Forensics and Security*, 4(4):638–650, Dec. 2009.

[3] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Proceedings of the International Cryptology Conference (CRYPTO)*, Aug. 2010.

[4] V. Gratzer and D. Naccache. Alien vs. quine, the vanishing circuit and other tales from the industry's crypt. In *Proceedings of Eurocrypt*, May 2006.

[5] D. Grawrock. *Dynamics of a Trusted Platform: A Building Block Approach*. Intel Press, 2008.

[6] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual: Combined volumes:1, 2a, 2b, 2c, 3a, 3b, and 3c, Mar. 2012.

[7] M. Jakobsson and K.-A. Johansson. Assured detection of malware with applications to mobile platforms. In *HotSec*, Aug. 2010.

[8] B. Kauer. OSLO: Improving the security of Trusted Computing. In *Proc. USENIX Security*, 2007.

[9] R. Kennell and L. H. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of the 12th USENIX Security Symposium*, Aug. 2003.

[10] Y. Li, J. M. McCune, and A. Perrig. Viper: Verifying the integrity of peripherals' firmware. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[11] Y. Lu, L.-T. Lo, G. R. Watson, and R. G. Minnich. Car: Using cache as ram in linuxbios. `coreboot.org`, Sep 2006.

[12] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010.

[13] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proc. ACM European Conference in Computer Systems (EuroSys)*, 2008.

[14] T. Müller, A. Dewald, and F. Freiling. TRESOR runs encryption securely outside RAM. In *Proceedings of USENIX Security Symposium*, 2011.

[15] Rochester Electronics. Counterfeit electronic components in the U.S. supply chain. The Office of Technology Evaluation, U.S. Dept. of Commerce, 2010.

[16] A. Seshadri, M. Luk, and A. Perrig. SAKE: Software attestation for key establishment in sensor networks. In *Proceedings of International Conference on Distributed Computing in Sensor Systems (DCOSS)*, June 2008.

[17] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. VanDoorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proc. Symposium on Operating Systems Principals*, 2005.

[18] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: Software-based attestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.

[19] M. Shaneck, K. Mahadevan, V. Kher, and Y. Kim. Remote software-based attestation for wireless sensors. In *Proceedings of ESAS*, 2005.

[20] E. R. Sparks. A security assessment of trusted platform modules. Technical Report TR2007-597, Dartmouth College, June 2007.

[21] D. Spinellis. Reflection as a mechanism for software integrity verification. *ACM Transactions on Information and System Security*, 3(1):51–62, Feb. 2000.

[22] G. Suh, C. ODonnell, I. Sachdev, and S. Devadas. Design and implementation of the AEGIS single-chip secure processor using physical random functions. In *Proceedings of Annual International Symposium on Computer Architecture (ISCA)*, June 2005.

[23] A. Triulzi. The Jedi Packet takes over the Deathstar, taking NIC backdoor to the next level. In *The 12th annual CanSecWest conference*, 2010.

[24] VSI Alliance. The value and management of intellectual assets. `http://vsi.org/documents/datasheets/TOC20IPPWP210.pdf`, 2002.

[25] E. Witchel. *Mondriaan Memory Protection*. PhD thesis, Massachusetts Institute of Technology, 2004.

[26] A. C. Yao. Protocols for secure computations. In *Proceedings of 23rd Annual Symposium on Foundations of Computer Science*, Nov 1982.