By ARVIND SESHADRI, MARK LUK, ADRIAN PERRIG, LEENDERT VAN DOORN, and PRADEEP KHOSLA

EXTERNALLY VERIFIABLE CODE EXECUTION

Using hardware- and software-based techniques to realize a primitive for externally verifiable code execution.

Computing devices are routinely targeted by a wide variety of malware, such as spyware, trojans, rootkits, and viruses. The presence of exploitable vulnerabilities in system software, and the widespread availability of tools for constructing exploit code, has reduced the amount of effort required for attackers to introduce malware into computing devices. Increasing levels of network connectivity further exacerbates the problem of malware propagation by enabling attacks to be launched remotely. Current computing devices are routinely used for security-sensitive applications; thus malware present on these devices can potentially compromise the privacy and safety of users. Furthermore, most computing devices today are part of a large networked infrastructure. Hence, the compromise of any one computing device can lead to the compromise of the networked applications. For example, a rogue wireless LAN access point can modify network traffic, thereby potentially affecting all computing devices that use this access point. Therefore, to use computing devices with confidence, users need assurance the software on their own computing devices and other computing devices in their network executes untampered by malware.

In this article, we describe a new primitive called externally verifiable code execution. This primitive allows an external entity (the verifier) to obtain assurance that an arbitrary piece of code, called the target executable, executes untampered by any malware that may be present on an external computing device. Assuming the target executable is self-contained (does not invoke any other code) and does not contain any software vulnerabilities, externally verifiable code execution is equivalent to the following two guarantees:

- Correct invocation: The verifier obtains the guarantee that the correct target executable image is loaded into memory and invoked for execution.
- Untampered execution: Other than performing denial-of-service attacks, no malware that may exist on the computing device can interfere with the execution of the target executable in any manner.

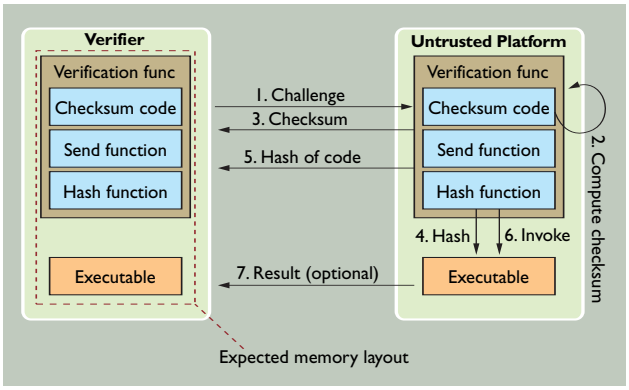

Figure 1. Overview of Pioneer. The numbers represent the temporal ordering of events.

Software-based and hardware-based techniques for externally verifiable code execution have been proposed. Both classes of techniques rely on a root of trust on the computing device. The root of trust is a trusted computing base responsible for enforcing externally verifiable code execution by performing three actions: measuring integrity of the target executable; setting up appropriate protections to isolate the target executable from all other software; and invoking the target executable for execution. The root of trust also sends the integrity measurement to the verifier over an authenticated communication channel. The verifier, which knows the correct value of the target executable's integrity measurement, uses the received integrity measurement to verify if the correct target executable was invoked for execution. In addition, since the execution of the target executable is isolated from all other software on the computing device, the verifier obtains the guarantee of untampered execution of the target executable.

Software-based and hardware-based techniques for externally verifiable code execution differ in how they establish the root of trust. Pioneer is a software-based technique in which the root of trust is established dynamically [3]. In Pioneer, the computing device executes a self-checksumming function called the verification function, which computes a checksum over its own instruction sequence. The verification function also sets up an isolated execution environment, wherein its execution is isolated from all other software on the computing device. This is achieved by first detecting if any other code had been executing concurrently. Then, we set up the execution environment to prevent other code from running in the future. The isolated execution environment guarantees that no malware on the computing device can interfere with the execution of the verification function. If the attacker modifies the verification function in any manner or fakes the creation of the isolated execution environment, the checksum computed by the verification will be incorrect. If the attacker tries to forge the correct checksum despite executing a modified verification function or an incorrectly created isolated execution environment, the time taken to compute the checksum will increase noticeably. Thus, the verifier obtains an assurance that the verification function on the computing device remains unmodified and the isolated execution environment had been properly set up if the checksum returned by the computing device is correct, and the checksum is returned within the expected amount of time.

When these two conditions hold, the verifier obtains the guarantee that a dynamically created root of trust exists on the computing device in the form of the verification function.

In the two hardware-based approaches for externally verifiable code execution, the Pacifica technology by AMD and the LaGrande technology (LT) by Intel, a subset of the computing device's hardware makes up the root of trust [1, 2]. Assuming the hardware remains uncompromised, the root of trust is always present on the computing device. This is generally a valid assumption, since historically, attackers prefer software-based attacks that are much easier to launch than hardware-based attacks.

Pioneer

Here, we provide an overview of Pioneer, which is a software-based primitive for externally verifiable code execution. We examine the assumptions and

the attacker model, describe the verification function, and explore the Pioneer challenge-response protocol.

Assumptions and Attacker Model. We assume the verifier knows the exact hardware configuration of the computing device, including the CPU model, the CPU clock speed, and the memory latency. In addition, the computing device has a single CPU. A trusted network is needed to eliminate the proxy attack, where the computing device asks a faster computing platform (proxy) to compute the checksum on its behalf. Thus, we assume a communication channel such that the verifier can detect any attempts by the computing device to contact other computing platforms.

We also assume the target executable is self-contained, and does not invoke any other software during its execution. Also, the target executable can execute at the highest CPU privilege level without generating exceptions and with all interrupts disabled.

We consider an attacker who has complete control over the software of the computing device. In other words, the attacker can tamper with all software, including the OS, and inject arbitrarily malicious software into the computing device. However, we assume the attacker does not modify the hardware. For example, the attacker does not load malicious firmware onto peripheral devices such as network cards or disk controllers, or replace the CPU with a faster one. In addition, the attacker does not perform Direct Memory Access (DMA) attacks such as scheduling a benign peripheral device to overwrite the verification function or the target executable with a DMA-write.

1.	V:	$t_1 \leftarrow$ current time, $nonce \xleftarrow{R} \{0,1\}^n$
	$V \rightarrow P$:	$\langle nonce \rangle$
2.	P:	$c \leftarrow \text{Checksum}(nonce, P)$
3.	$P \rightarrow V$:	$\langle c \rangle$
	V:	$t_2 \leftarrow$ current time
		if $(t_2 - t_1 > \Delta t)$ then exit with failure
		else verify checksum c
4.	P:	$h \leftarrow \text{Hash}(nonce, E)$
5.	$P \rightarrow V$:	$\langle h \rangle$
	V:	verify measurement result h
6.	P:	transfer control to E
7.	$E \rightarrow V$:	$\langle$result (optional)$\rangle$

Figure 2. The Pioneer protocol. The numbering of events is the same as in Figure 1. *V* is the verifier, *P* is the verification function, and *E* is the executable.

Pioneer Overview. The verification function dynamically instantiates the root of trust on the computing device. As Figure 1 shows, the verification function consists of three parts: a checksum code, a send function, and a hash function. The checksum code computes the checksum over the instructions of the verification function and also sets up an isolated execution environment for the verification function. After the checksum code finishes computing the checksum, it invokes the send function to transfer the checksum to the verifier over the communication channel. After sending the checksum back to verifier, the checksum code invokes the hash function, which computes the integrity measurement of the target executable by computing a hash over the executable image. After returning the hash value to the verifier via the send function, the hash function invokes the target executable. Since the target executable is directly invoked by the hash function, which executes in the isolated execution environment set up by the checksum code, the target executable inherits the same isolated execution environment. Furthermore, by assuming the target executable to be self-contained, we arrive at the guarantee that no malware on the computing device can affect the execution of the target executable.

The checksum code computes a "fingerprint" of the verification function such that the checksum will be different even if one byte of this region had been modified. Tampering with the checksum code to forge the correct checksum would induce a time delay for the attacker. Therefore, Pioneer requires a time-optimal implementation of the checksum code; otherwise an attacker could replace the Pioneer checksum code with a faster, malicious implementation and still generate the correct checksum. To aim for time-optimality, we designed the Pioneer checksum code to be a short sequence of simple arithmetic and logical assembly instructions.

The isolated execution environment enforces the atomicity of execution of the verification function and the target executable. That is, no other code on the computing device is allowed to execute until the verification function and the target executable have finished executing. This atomicity is achieved by ensuring that the verification function and the target executable execute at the highest CPU privilege level with all maskable interrupts disabled. Also, as part of the checksum computation process, new handlers that are part of the verification function image are installed for all non-maskable interrupts and exceptions. Since the target executable is self-contained, no code other than the verification function or the target executable can execute as long as either of these two entities is executing.

Pioneer is based on a challenge-response protocol between the verifier and the computing device. Figure 2 shows the Pioneer protocol. The verifier invokes the verification function on the computing device by

sending a random nonce as the challenge. The checksum code in the verification function computes the checksum over the verification function's instruction sequence as a function of the random nonce. Using a long random nonce prevents pre-computation and replay attacks. The checksum code sends the computed checksum back to the verifier using the send function. The verifier has a copy of the verification function and can hence independently compute the checksum. The verifier confirms that the checksum returned by the computing device is correct and the checksum is returned within the expected amount of time. If these two conditions are met, the verifier obtains the guarantee that the root of trust has been correctly instantiated on the computing device.

1. C: execute skinit
2. C → T: read E from memory, send over LPC bus
3. T: $PCR_i \leftarrow$ Hash (PCR_i, E)
4. C: execute E
5. V → T: ? ⟨nonce⟩
6. T: $\sigma \leftarrow \{PCR_i, nonce\}_{AIK^{-1}}$
7. T → V: ? $\langle PCR_i, \sigma \rangle$

Figure 3. Pacifica-based externally verifiable code execution.

After sending the checksum back to the verifier, the checksum code invokes the hash function, which computes a hash of the target executable image concatenated with the random nonce sent by the verifier. The hash function then sends the hash of the target executable back to the verifier. The verifier verifies the correctness of the hash value using its own copy of the target executable image. Finally, the hash function invokes the target executable for execution. Note that if the checksum returned by the computing device was successfully verified, the verifier knows that the hash function remains unmodified and executes in an isolated execution environment. Hence, by receiving the correct hash value, the verifier obtains the guarantee that the correct hash function must be executing, instead of a malicious hash function that hashes the correct target executable but invokes a malicious program.

The final outcome of the Pioneer protocol guarantees to the verifier that the correct code had been invoked inside an untampered execution environment. Many details had been omitted here, but can be found in [3].

Pacifica and LT

Two hardware-based approaches for externally verifiable code execution are examined here: AMD's Pacifica and Intel's LT.

Assumptions and Attacker Model. Similar to Pioneer, we assume the target executable is self-contained, and can execute at the highest CPU privilege level without generating exceptions and with all interrupts disabled. Also, we consider an attacker who can arbitrary modify the software of the computing device.

Overview of Pacifica and LT. We now describe how Pacifica and LT can be used to obtain the guarantee of externally verifiable code execution. We base our discussion on Pacifica since the technical details of LT are presently unavailable to the public.

Both LT and Pacifica require a Trusted Platform Module (TPM) to be present on the computing device to provide the guarantee of externally verifiable code execution. The TPM is a tamper-resistant cryptographic coprocessor based on standards released by the Trusted Computing Group (TCG) [4]. The TPM computes integrity measurements of program images using the SHA-1 cryptographic hash function. The TPM stores these integrity measurements in protected registers called the Platform Configuration Registers (PCR). When requested by an external verifier, the TPM returns one or more of the PCR values digitally signed using the private half of its Attestation Identity Key (AIK). The TCG standards for the TPM describe in detail how the AIK key pair is securely generated inside the TPM, and how a trusted third party can verify the public half of the AIK belongs to a particular TPM on a particular computing device. The verifier has a certificate containing the public half of the AIK, and uses this certificate to authenticate incoming PCR values. This certificate can also be used to detect if malware had modified the encrypted private half of the AIK stored outside the TPM. The verifier can then use the integrity measurements contained in the PCR values to determine what software had been loaded on the computing device. This process by which an external verifier can determine what software is present on a computing device is called attestation.

To enable externally verifiable code execution using a TPM, Pacifica adds a special instruction called `skinit` (the corresponding LT instruction is called `senter`) to the CPU's instruction set. This instruction takes a pointer to the target executable as its operand. When executing a `skinit` operation, the CPU reinitializes itself without resetting any of the peripherals or the memory. The CPU then transfers

the entire target executable image (which must be less than 64Kb in size) to the TPM using the dedicated Low-Pin-Count (LPC) bus in a manner that cannot be simulated by software. Thereby, an attacker cannot fake the execution of `skinit` in software. The TPM hashes the target executable image and stores the hash in a PCR. After transferring the target executable image to the TPM, the CPU sets up an isolated execution environment for the target executable by disabling all interrupts and setting up memory protections to disable DMA writes to the memory locations containing the target executable. Finally, the CPU jumps to the target executable and executes it. An external verifier can read the PCR value containing the integrity measurement of the target executable by making an attestation request to the TPM. Since the verifier has a copy of the target executable image it can verify the correctness of the integrity measurement. As with Pioneer, the verifier is assured that any malware on the computing device cannot tamper with the execution of the target executable due to the correctness of the integrity measurement and the fact the target executable executes in an isolated execution environment instantiated by the CPU. Figure 3 shows the sequence of events in Pacifica-based externally verifiable code execution.

Comparing Pioneer and Pacifica/LT

The primary advantage of Pioneer is that being software-based, it can be used with legacy computing devices as well as computing devices that lack the hardware support necessary for LT and Pacifica. Also, Pioneer can be easily updated when vulnerabilities are discovered. Updating LT or Pacifica will be difficult because they are hardware-based, as an example will illustrate. The software running on TPMs needs to be updated in light of the recent compromise of the collision resistance property of SHA-1 hash function [5]. However, TPMs are designed so that their software is not field-upgradable to prevent attackers from exploiting the update facility. Therefore, the only way to perform the hash function update is to physically replace the TPM chip.

On the other hand, Pioneer has some drawbacks compared to LT and Pacifica. Pioneer requires an authenticated communication channel between the verifier and the computing device. Therefore, Pioneer cannot be used when the verifier and the computing device communicate over an untrusted network like the Internet. Also, Pioneer has some open research issues. In particular, we need to prove the time-optimality of the checksum code, prove the absence of mathematical optimizations to the checksum code, and derive a checksum code that can be easily ported across different CPU architectures (the current version of Pioneer is designed for the x86_64 architecture). There are also low-level attacks that need to be addressed: CPU overclocking, DMA-writes by peripherals, memory writes by malicious CPUs in a multi-CPU system, and variations in the running time of the checksum code due to dynamic CPU clock scaling.

Conclusion

We have described the primitive of externally verifiable code execution and given an overview of software-based and hardware-based techniques to realize this primitive. Externally verifiable code execution can be used as a building block to construct security applications that will empower users to use their computing devices with confidence in the face of the ever-increasing threat of malware. We hope this article will motivate other practitioners in the field to embrace this technology, extend it, and apply it to build secure systems.

References

1. *AMD Corp. AMD64 Architecture Programmer's Reference Manual* (Dec. 2005).
2. *Intel Corp. LaGrande Technology Architectural Overview*. (Sept. 2003).
3. Seshadri, A. et al. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, (Oct. 2005), 1–15.
4. Trusted Computing Group (TCG). *TCG TPM Specification Version 1.2*, (Mar. 2006).
5. Wang, X., Yin, Y. and Yu, H. Finding collisions in the full SHA-1. In *Proceedings of Crypto*, (Aug. 2005).

Arvind Seshadri (arvinds@cs.cmu.edu) is a Ph.D. candidate in the Department of Electrical and Computer Engineering at Carnegie Mellon University in Pittsburgh, PA.

Mark Luk (mluk@ece.cmu.edu) is a research scientist at CyLab at Carnegie Mellon University in Pittsburgh, PA.

Adrian Perrig (perrig@cmu.edu) is an assistant professor of Electrical and Computer Engineering, Engineering and Public Policy, and Computer Science at Carnegie Mellon University in Pittsburgh, PA.

Leendert van Doorn (leendert@us.ibm.com) is a senior manager of the Secure Systems and Tools Department at IBM T.J. Watson Research Center in Yorktown Heights, NY.

Pradeep Khosla (pkk@ece.cmu.edu) is the dean of Carnegie Institute of Technology at Carnegie Mellon University in Pittsburgh, PA.