Conqueror: tamper-proof code execution on legacy systems

Lorenzo Martignoni[†], Roberto Paleari[‡], and Danilo Bruschi[‡]

[†]Università degli Studi di Udine, [‡]Università degli Studi di Milano
lorenzo.martignoni@uniud.it, {roberto,bruschi}@security.dico.unimi.it

Abstract. We present Conqueror, a software-based attestation scheme for tamper-proof code execution on untrusted legacy systems. Beside providing load-time attestation of a piece of code, Conqueror also ensures run-time integrity. Conqueror constitutes a valid alternative to trusted computing platforms, for systems lacking specialized hardware for attestation. We implemented a prototype, specific for the Intel x86 architecture, and evaluated the proposed scheme. Our evaluation showed that, compared to competitors, Conqueror is resistant to static and dynamic attacks and that our scheme represents an important building block for realizing new security systems.

1 Introduction

Code attestation is the process of verifying the integrity of a piece of code executing in an untrusted system. Besides integrity verification, code attestation can also be used to execute an arbitrary piece of code in an untrusted system with the guarantee that the code is run unmodified and in an untampered execution environment. In the last years, hardware extensions, such as TPM chips [1], have been proposed for securing computations, including performing attestation. However, these extensions are not yet available on every computing device. In such a situation, pure software-based solutions are the only viable alternative.

Several software-based attestation schemes have been proposed in literature [2–7]. All these schemes are based on a challenge-response protocol involving two parties: an *untrusted system* and a *verifier*. The verifier issues a challenge for the untrusted system, where the challenge consists in computing the checksum of certain memory locations and properties of the execution environment. The checksum is computed by executing a particular *attestation routine*, or *checksum function*. Once computed, the checksum is sent back to the verifier. The verifier relies on the time to determine whether the checksum is genuine or if it could have been forged. Indeed, attestation routines are constructed such that any tampering attempt results in a noticeable increase of the execution time. Thus, a checksum received too late is a symptom of an attack.

The complexity of the attestation routine depends on the hardware characteristics of the untrusted system on which it has to be executed. Indeed, the output of the routine is guaranteed to be genuine only if it is executed in a properly configured execution environment. In complex hardware architectures, such

as the ones used in personal computers, there exist several configurations of the execution environment that can be exploited by an attacker to thwart attestation. Therefore, the attestation routine must ensure, and prove to the verifier, that the execution environment in which it executes satisfies all the requirements to impede attacks. In other words, the attestation routine must attest its own code, but also the execution environment. Intuitively, the requirements for tamper-proof attestation are that the attestation routine must be executed at the highest level of privilege (i.e., at the same level of the most powerful attacker) and that its execution must be uninterruptible. Practically speaking, in a legacy system with no hardware support for virtualization, that means that the routine must execute in system mode (i.e., the privilege level of the operating system) and that all interrupts must be disabled, to prevent the attacker to regain the control of the execution at some point. Unfortunately, even if the requirements are very well defined, guaranteeing that they are satisfied in a complex execution environment where attacker and defender have the same privileges is a very challenging problem.

In this paper we present Conqueror, a software-based scheme for tamper-proof code execution on untrusted legacy systems. Conqueror provides a security primitive that allows to build applications that require the availability of a trusted computing base. Pragmatically speaking, Conqueror guarantees that an arbitrary piece of code can be executed untampered in an untrusted system, even in the presence of malicious software. Conqueror has been developed to address the limitations of Pioneer, the state-of-the-art software-based attestation solution [6]: Conqueror is immune to all attacks that are known to defeat Pioneer, and it can also be used on untrusted systems where the attacker could leverage hardware virtualization extensions to hold control of the execution environment in which the attestation routine executes. Conqueror adopts a variation of the challenge-response protocol used in traditional attestation schemes: the challenge does not consist in a seed to initialize a constant attestation routine, but instead consists in an entire routine, that is different each time, self-decrypting, and obfuscated. The intent is to make it impossible for an attacker to reverse engineer the logic of the checksum computation, and to facilitate the hiding of the sensitive operations that Conqueror needs to perform to attest that the state of the environment executing the code impedes any attack. The strength of this approach is that we are drastically increasing the time needed by an attacker to forge a checksum.

We experimentally demonstrate our claims about Conqueror's resistance to attacks. We show that even a preliminary low-level analysis of the code of Conqueror's one-time attestation routine (i.e., disassembly), which is necessary to perform any subsequent meaningful analysis for reconstructing the semantics, costs about the same time required to execute the routine. Moreover, we show that Conqueror is also resilient to dynamic attacks performed by an attacker leveraging a hardware-assisted hypervisor. Finally, to demonstrate Conqueror's potential, we present a proof-of-concept software-based primitive to launch securely a hypervisor in a running untrusted system, to segregate the system into a restricted guest. This primitive could be used in place of `skinit` [8] and

`senter` [1] on untrusted systems with no hardware support for trusted computing.

2 State-of-the-art of attestation on legacy systems

This section presents Pioneer, the major Conqueror's competitor. Both systems target the same hardware architecture, but they use very different approaches. Moreover, Conqueror is resistant to attacks that are known to defeat Pioneer.

Pioneer is a software-based attestation scheme that can be used to establish a trusted computing base, called *dynamic root of trust*, on an untrusted legacy system. Pioneer is specific for Intel x86 with EM64T extensions. The code of the dynamic root of trust is guaranteed to be unmodified and to execute in a *tamper-proof execution environment*. The dynamic root of trust measures the integrity of an arbitrary executable, and then runs the executable in the trusted execution environment. The dynamic root of trust is established using a *verification function*. The verification function is an extension of a conventional checksum function and additionally includes a hash function to verify the integrity of an executable. The verification function is self-checking (i.e., it attests its own code), and it attests the execution environment.

The Pioneer verification function is composed by three components: (I) a checksum function, (II) a send function, and (III) a hash function. The checksum function is used to compute a checksum over the entire verification function and to setup the execution environment in which the other functions are guaranteed to run untampered. Since the sensitive component of Pioneer is the checksum function, we do not overview the others.

As in the majority of code-attestation schemes, in Pioneer the checksum function is known a priori and the challenge issued by the verifier consists in a seed that initializes this function. Therefore, an attacker has complete access to the checksum function and can analyze it offline to find weaknesses. The checksum function has been constructed manually to be time-optimal: no adversary function that can compute the correct checksum without introducing a noticeable overhead exists. Time-optimality is achieved using operations that prevent parallelization, that have a low variance execution time, and by executing these operations iteratively, to maximize the overhead of the attacker. Most importantly, the checksum function is responsible for initializing the execution environment and for attesting the correct initialization.

Unfortunately, since the hardware architecture for which Pioneer was developed is full of subtle details, researchers have found ways to thwart the setup of the dynamic root of trust without being noticed by the verifier. For example, it is possible to perform the entire checksum computation in user-space and to regain the control of the execution through exceptions without corrupting the checksum. Another attack consists in desynchronizing data and code pointers and to execute a modified checksum function that computes the checksum of a pristine function residing elsewhere in memory [9]. Finally, Pioneer's assumptions that the most powerful attacker operates in system mode does not hold on new commodity hardware with support for virtualization [8, 10].

3 Conqueror overview

In this section we give an overview of Conqueror, our scheme for software-based code attestation and tamper-proof code execution on untrusted legacy systems (Intel x86). Conqueror does not suffer the problems that affect the state-of-the-art attestation scheme for this class of systems.

3.1 Threat model

Conqueror has been developed to operate in the following adversary scenario. We assume that the untrusted system has been compromised, and that the attacker operates at the highest privilege level: system mode (ring 0) if the system has no support for hardware-based virtualization, hypervisor mode if the support is available. However, we assume the adversary cannot operate in system management mode, that he cannot perform hardware-based attacks (e.g., DMA-based attacks or overclocking), and that he cannot leverage a pristine or a more powerful system to break the attestation scheme. The final assumption is that the untrusted system supports a single thread of execution (e.g., no SMP).

3.2 Conqueror architecture and protocol

As any other software-based code attestation scheme, Conqueror is based on a challenge-response protocol, where a verifier challenges the untrusted system. The central component of Conqueror is the Tamper-Proof Environment Bootstrapper (TPEB). As the name says, the TPEB is responsible for setting up the environment in the untrusted system for the tamper-proof execution of an arbitrary executable. Figure 1 shows the layout and the protocol of Conqueror (the numbers in the figure represent the temporal ordering of the events). The TPEB is composed by a checksum function and a send function. The checksum function computes the checksum to attest the integrity of the TPEB itself and the integrity of the executable. The send function transmits the computed checksum value to the verifier and invokes the executable. The send function is logically separated from the checksum function because it is hardware dependent (i.e., it depends on the network card installed on the untrusted system).

In Conqueror the verifier generates the checksum function on demand, such that each function differs considerably from the others. Differences are both syntactic and semantic. Moreover, functions are obfuscated using multiple obfuscation schemes. The attacker has no access to the checksum function ahead of time and cannot perform any offline analysis nor optimization [7]. In Conqueror, the newly generated checksum function is initially sent encrypted to the untrusted system. Later on, at time t_0, the verifier transmits the key for decryption. Since the verifier knows precisely in which execution environment the function must be executed and knows the hardware characteristics of the untrusted system, it can compute the expected checksum value and can estimate the amount of time that will be required by the untrusted system to decrypt, to execute the function, and to send back the result. Let $t_1 = t_0 + \Delta_t$ be the time by which the

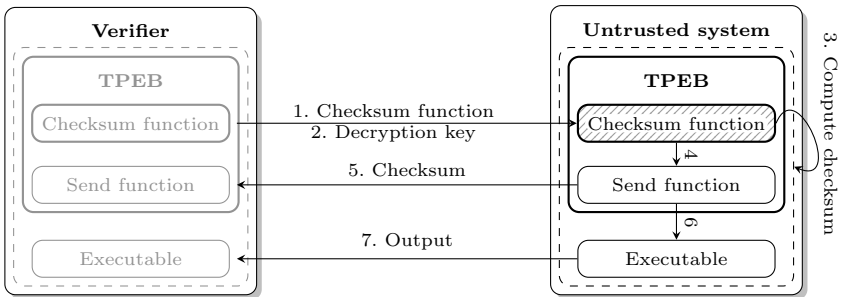

Fig. 1. Overview of Conqueror

correct checksum has to be received by the verifier to be considered authentic; Δ_t is an upper bound, empirically estimated, of the maximum time requested by the untrusted system to compute the checksum in the absence of an attack. If the verifier does not receive the correct checksum by t_1, then the checksum is considered forged and the execution environment not tamper-proof. In a traditional checksum function (e.g., that used in Pioneer), where the function is known a priori and can be analyzed offline, the attacker has Δ_t time to execute a malicious function to forge the checksum. In Conqueror, the attacker has Δ_t to (I) analyze the checksum function, (II) generate a new function capable of forging the checksum, and (III) execute the generated function. Alternatively, the attacker would have to emulate the entire execution of the checksum function. Differently from traditional checksum functions, the ones in Conqueror are generated automatically; for this reason we cannot guarantee a low collision rate nor that their implementation is optimal (in terms of execution time and in code size). Nevertheless, given the small time frame available, there is no opportunity for the attacker to reverse engineer their semantics, nor to emulate the execution, and to forge checksums in time.

Since Conqueror targets a very complex hardware architecture, particular attention has to be devoted to prevent checksum forgery, by tampering either the checksum function or the execution environment. To attest the trustworthiness of the environment, the verifier embeds in the checksum function several operations whose behavior and execution time depend on the configuration of the environment (e.g., instructions that raise exceptions when executed without enough privileges).

An attacker who tampers the execution of the checksum function will corrupt the checksum, or will incur in a time overhead that will cause the overall checksum computation to exceed the expected time Δ_t. For these reasons, Conqueror guarantees that a correct checksum, received by the verifier by t_1, is the proof that the checksum function has been executed unmodified and that the bootstrap of the tamper-proof execution environment succeeded.

4 Conqueror implementation

Conqueror current implementation is specific for the Intel x86 architecture and so are the details of the implementation presented in this section. However, we believe the same scheme can be used, as is, on the Intel x86-64 architecture.

4.1 Tamper-Proof Environment Bootstrapper

The layout in memory of the TPEB is shown in Figure 2. The TPEB consists of the checksum function, its data, and the send function. For simplicity, the TPEB is located at a fixed address (`BASE`) and in consecutive memory pages. Moreover, the executable follows immediately the TPEB, and the overall buffer is padded to a multiple of page size (`SIZE`). We assume that the TPEB is already initialized on the untrusted system, with the exception of the checksum function. The checksum function and its data reside in a dedicated memory page (starting from `BASE`) and all unused bytes in this page are initialized randomly, to hide code and data. This page is generated on-demand by the verifier and transmitted encrypted to the untrusted system. The latter stores in memory, at the `BASE` address, the page and waits for the decryption key. Attestation begins when the verifier sends out the key. The reason for encryption is to exclude from the measurement the time required to transmit and prepare the TPEB.

4.2 Checksum function

The checksum function is composed by a prologue, a checksum loop, and an epilogue (Figure 2). The prologue decrypts the rest of the page containing the checksum function, initializes the execution environment for the remaining of the computation, and invokes the checksum loop. The checksum loop (described in Section 4.2) computes the checksum of the memory pages containing its own code, the send function, and the executable, (i.e., from `BASE` to `BASE + SIZE`), and invokes the epilogue. The epilogue invokes the send function, which in turn invokes the executable.

The checksum function computes the checksum by combining multiple checksum gadgets. In the current implementation the checksum size is 128 bits. A *gadget* (c_i) is a small code snippet that receives in input the address of a memory location and updates the running value of the checksum, according to the content of the memory. We refer to these gadgets as *active*, since they are intentionally executed by the checksum function. The purpose of an active gadget is twofold. First, each gadget contributes to the computation of the checksum in a different way. Thus, the correct checksum can be computed only if all the gadgets are executed in the proper order and with the proper arguments. Second, certain gadgets perform additional operations to verify the trustworthiness of the execution environment and, in case the environment has been tampered, they either corrupt the checksum or introduce a time overhead. Since gadgets are scattered around the memory, differ syntactically and semantically from one checksum function to another, and are obfuscated, it becomes very difficult for the attacker to reconstruct the exact logic of the checksum function.

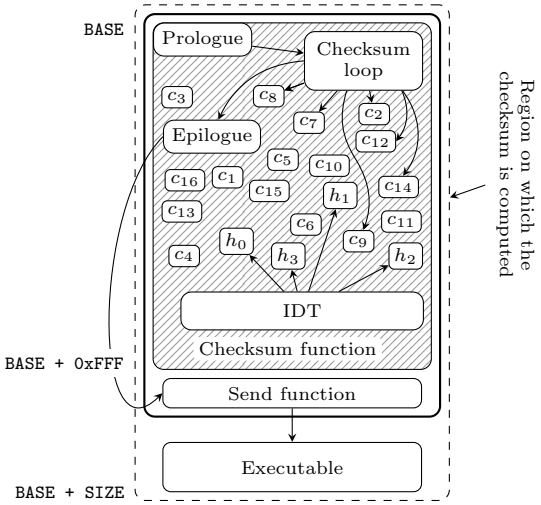

Fig. 2. Overview of the TPEB

In addition to active gadgets, the checksum function relays on *passive gadgets* (h_j), or *handlers*, that are not invoked directly by the checksum function, but rather as the result of an unexpected event that can occur only in a tampered execution environment. If executed, passive gadgets corrupt the checksum. Passive gadgets are registered during the prologue, by replacing the Interrupt Descriptor Table (IDT) with a new one embedded within the TPEB, and cannot be disabled by the attacker: an improper configuration of these gadgets will result in a wrong checksum.

Prologue. The prologue (Figure 3) is a small routine that decrypts the rest of the page and initializes the trusted execution environment. More precisely, the prologue disables all maskable interrupts (line 2), decrypts the rest of the page (line 4 and 5), and installs custom interrupts handlers (line 7). Custom handlers are installed by updating the address of the interrupt descriptor table (IDT). The new address is set to a location, within the memory page containing the checksum function, that holds a pre-initialized IDT (Figure 2). The mapping between interrupts and handlers (the content of the IDT) is chosen by the verifier and not known to the attacker. The handlers (h_i in Figure 2), or passive gadgets, are a special type of gadget: like normal gadgets they modify the running value of the checksum, but they terminate their execution with a special instruction to return to normal execution (i.e., `iret`). Furthermore, handlers are never invoked explicitly by the checksum loop but only in response to interrupts or exceptions.

The purpose of the prologue is twofold. First, by disabling maskable interrupts (pin-based interrupts generated by the peripherals) we inhibit the asynchronous execution of all handlers. Second, by installing custom interrupt handlers that update the checksum value, we can tell whether any interrupt or ex-

ception occurred during the computation of the checksum. If maskable interrupts are successfully disabled, no asynchronous interrupt occurs, and the checksum is not corrupted because no interrupt handler is fired. Similarly, if the checksum loop executes privileged instructions, and the checksum function is executed in system mode, no exception occurs and no exception handler corrupts the checksum. On the other hand, any attempt to execute the checksum function in user mode results in an exception, in the execution of the corresponding handler, and in a corruption of the checksum value.

By positioning the IDT in the same memory page of the checksum function, we implicitly certify the content of the table. The only opportunity for the attacker is to intercept and simulate a successful update of the IDT. For example, the attacker could emulate the execution of the prologue or execute the prologue in user-space, such that the update of the IDT will raise an exception and will be intercepted. Then, the attacker could install his own malicious IDT and simulate a successful disabling of maskable interrupts. We prevent this attack by including in the checksum loop a special gadget that queries the address of the IDT and updates the running value of the checksum accordingly. Therefore, attacker's attempts to relocate the IDT will result in a corrupted checksum. Further details about the aforementioned gadget and about why its execution cannot be detected by the attacker are given in Section 4.2.

In conclusion, a correct value of the checksum, received by the verifiers within the expected time, certifies that the prologue is executed successfully, that the checksum function is executed at the maximum privilege level, and that the attacker cannot interrupt the execution using interrupts or exceptions.

Checksum loop. The core of the checksum computation is the checksum loop shown in Figure 4. The checksum loop is composed by two nested loops. The innermost loop traverses the memory and updates the checksum according to the content of the memory, invoking a different gadget at each iteration. The memory is not traversed linearly, but instead in a pseudorandom fashion (line 4), using a T-function [11]. The T-function produces a pseudorandom permutation of all the memory locations to traverse. More precisely, the T-function returns the memory offset of the next memory location for the checksum computation. At each iteration (line 5), from the offset returned by the T-function, the checksum loop computes the absolute address of the memory location to process, and invokes a specific gadget to update the running value of the checksum (`GADGETS` represents the number of gadgets available). Clearly, without an analysis of the code, the attacker cannot predict which gadgets will process which memory locations and, even if the checksum function were weak (e.g., it suffers a high collision rate), the attacker would not have enough time to exploit the weakness. Finally, it should be noted that the execution of the checksum loop is deterministic, unless it is tampered.

The outermost loop repeats the memory traversal multiple times (`ITERATIONS` denote the number of iterations of the outermost loop). At each iteration, the T-function used in the innermost loop is initialized with a different seed (line 2). Therefore, the innermost loop is executed multiple times and at each execution

```
1 // Disable maskable interrupts
2 asm("cli");
3 // Decrypt the remaining of the page
4 for (i = PROLOGUE_SIZE; i < 4096; i++)
5   BASE[i] ^= KEY[i % KEY_SIZE]
6 // Install custom interrupt handlers
7 asm("lidt %0" : : "m" (IDT));
```

Fig. 3. Overview of the prologue

```
1 for (i = 0, j = 0; i < ITERATIONS; i++) {
2   x = seed(i) % (SIZE / 4);
3   do {
4     x = (x + (x*x | 5)) % (SIZE / 4);
5     checksum_gadget[j++ % GADGETS](BASE + x*4);
6   } while (x != seed(i) % (SIZE / 4));
7 }
```

Fig. 4. Overview of the checksum loop (in C for clarity)

the running value of the checksum is updated using a different combination of memory locations and gadgets, and the order in which the checksum is updated is also different. Since the checksum function is constructed such that any attacker's attempt to forge the correct checksum will introduce an overhead in the computation of the checksum, the outermost loop causes a constant time overhead per iteration and facilitates the detection of the attack. Details about how we select the optimal number of iterations for the outermost loop are given in Section 5.

The seeds used by the T-function to generate the addresses are also included in the memory page containing the checksum function. To avoid wasting precious bytes of the page, the vector containing the seeds is positioned at a random location within the page and is not initialized, to overlap with the existing content of the page.

Checksum gadgets. The checksum is computed by executing a sequence of gadgets, each of which contributes to update the running value of the checksum in a different way. Certain gadgets also perform additional operations to attest the trustworthiness of the execution environment. Given that gadgets are very small in size and that an entire memory page is dedicated to the checksum function, the checksum function can rely on about a hundred different gadgets simultaneously. Gadgets are generated on demand by the verifier and change (in number, position, syntax, and semantics) from challenge to challenge.

The following paragraphs describe in details the gadgets used in the checksum function to attest the integrity of the TPEB and of the code of the executable. Figure 5 shows some sample gadgets. For clarity, the gadgets presented are not optimized and use symbolic names (in uppercase) to refer to absolute memory locations containing data: `CHKSUM` and `ADDR` refer respectively to the memory locations storing the 128-bit checksum and the address of the next word to process.

Plain checksum computation. The simplest and most frequently used gadget is responsible only for updating the running value of the checksum. Different gadgets update the checksum in different ways, by applying different arithmetical or logical operations and by modifying different bits of the checksum value. Figure 5(a) shows a sample gadget. The gadget updates the checksum by adding the result of a bitwise XOR between the current memory location (`ADDR`) and a

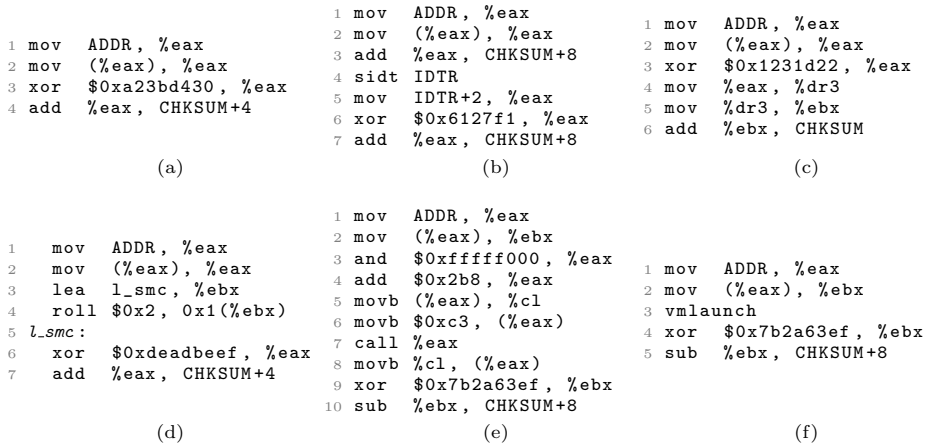

Fig. 5. Sample gadgets for (a) plain checksum computation, (b) IDT attestation, (c) system mode attestation, (d,e) instruction and data pointers attestation, and (f) hypervisor detection.

random key (`0xa23bd430`). Note that this gadget modifies the second word of the running 128-bit checksum (`CHKSUM+4`, at line 4).

IDT attestation. During the prologue, the interrupt descriptor table is replaced with a custom table, which is provided along with the checksum function. Since the prologue is executed at the beginning of the checksum function, it is reasonable to expect the attacker to try to emulate or intercept its execution.

The content of the IDT is implicitly attested by the normal checksum computation, but the address of the IDT is not. To attest that the IDT shipped with the checksum function is actually being used, the checksum function relies on a specific gadget that queries the CPU to obtain the address of the IDT and updates the checksum accordingly. Obviously, the checksum will be wrong if a different IDT is being used. The only opportunity for the attacker to force the checksum function to behave as if the requested IDT were successfully installed is to intercept the query and to manipulate its output. To query the address of the IDT, the gadget uses the `sidt` instruction. Unfortunately for the attacker, this instruction is not privileged: it does not trigger an exception when executed in user mode [12]. Consequently, the only solution for the attacker to detect the instruction is to analyze the checksum function or to emulate its execution. However, any analysis or emulation attempt will introduce a noticeable overhead in the computation of the checksum. Figure 5(b) shows a sample gadget to attest the IDT. The only difference with a plain gadget (Figure 5(a)) is the addition of the instructions to query the address of the IDT (lines 4 and 5).

System mode attestation. After the update of the IDT, the attacker cannot regain the control of the execution, because all interrupts and exceptions will be served by the handlers installed by the checksum function. Although the previously

described gadget forces the attacker to install our IDT, he could still attempt to execute the entire checksum function in user mode. If no maskable interrupt occurred during the execution of the checksum function, the checksum would not get corrupted, and the attack would not be detected. However, even if we suppose that the attacker executed the checksum function in user mode and that he were able to reprogram the interrupt controller to prevent any interrupt, he would lose any opportunity to regain the control of the system after checksum computation.

To have the guarantee that the TPEB is operating in system mode, the checksum function relies on a specific class of gadgets. These gadgets use a privileged instruction to update the running value of the checksum. If the function is executed in system mode, all the instructions of the gadgets will be executed successfully. However, if the function is executed in user mode, the privileged instruction will raise an exception (because of the lack of privileges), and the exception handler we installed to handle the exception will corrupt the checksum. In some cases, the handler could also trigger an endless loop. An example of such a gadget is shown in Figure 5(c). The gadget uses the CPU register `dr3` to store an intermediate result during the computation of the new checksum value. This register can be accessed only in system mode and any access originating from user mode causes a general protection fault exception.

Instruction and data pointers attestation. The checksum function is a self-checksumming function. A common class of attacks against self-checksumming functions are *memory copy attacks*, that allow attackers to forge checksums [6]. Briefly, in a memory copy attack, the attacker modifies the instructions of the checksum function, or the execution environment, to redirect all memory reads to memory locations containing a pristine copy of the data to attest. A memory copy attack can be performed in different ways: (I) by patching the instructions of the checksum function to read from different locations, (II) by configuring segmentation to separate the code from the data segment, and (III) by desynchronizing the data and the instruction TLBs [9].

To prevent memory copy attacks, the checksum function uses a specific type of gadget that guarantees that reads, writes, and fetches involving the same virtual memory location refer to the same physical location. Indeed, data and instruction physical pointers equivalence is sufficient to guarantee that no memory copy attacks of type (II) and (III) can be performed. We intentionally do not consider the case of memory copy attacks of type (I), performed by patching or by emulating the checksum function, because of the noticeable time overhead the attacker would suffer. To validate the equivalence of data and instruction pointers we leverage a gadget based on self-modifying code [13]. The gadget updates the running value of the checksum by performing an operation that is generated dynamically by modifying the code of the checksum function in place. If no memory copy attack is being performed, the data pointer (used for both reads and writes) and the instruction pointer point to the same physical page. Thus, the memory write executed by the gadget to update its instruction modifies the physical page that is also being executed. If the attacker were

performing a memory copy attack, the data and the instruction pointer would point to two different physical pages and the instruction executed to update the checksum would differ from the ones just created by the gadget. Consequently, the out-of-date instruction would corrupt the checksum.

Figure 5(d) shows a sample gadget used by Conqueror to prevent memory copy attacks. The gadget updates the checksum by adding the data read from the memory (lines 1, 2, and 7). Before the addition, the word read is XORed with an immediate (line 6). The immediate is rotated (by two bits) at each execution of the gadget, by modifying the operand of the instruction in place (line 3 and 4). In the case of a memory copy attack the checksum would not be updated correctly because the operand of the `xor` instruction would remain unmodified.

Note that, in the case of a memory copy attack of type (III), the attacker can operate on each page separately. The aforementioned gadget successfully protects against the desynchronization of data and instruction pointers that point to the page containing the checksum function, but, as is, it is ineffective at protecting other pages (containing the send function and the executable). Indeed, only instructions residing in the page containing the checksum function are executed during the checksum computation. To address this problem, we use a variation of the original gadget, that places a temporary small snippet of code (e.g., a `ret` instruction) in a random position of the input page, executes the snippet, and restores the original content of the modified locations. Figure 5(e) shows an example of this type of gadget. The gadget selects a random location in the page being attested (lines 1 to 4), saves the content of the location (line 5), replaces the content with a `ret` instruction (line 6), executes the newly generated instruction (line 7), restores the original content of the modified location (line 8), and finally updates the checksum (line 9 and 10).

Hypervisor detection. An attacker operating in hypervisor mode, on a system with hardware support for virtualization, has complete control of the operating system: he can intercept the execution of all sensitive instructions, interrupts, exceptions, and, most importantly, the hypervisor and the attacker are completely transparent to guests. Dai Zovi and Rutkowska *et al.* have clearly demonstrated what an attacker can do on systems with hardware support for virtualization [14, 15]. The gadgets presented so far are effective at attesting the trustworthiness of the execution environment only if we can guarantee that no attacker can operate in hypervisor mode. Therefore, the checksum function that attests the existence of a tamper-proof execution environment on the untrusted system must be adapted to compute the correct checksum value, in the expected amount of time, only when no hypervisor is running on the system.

There is a rich ongoing debate among researchers about hypervisors detection and hiding. Although, the hardware has been specifically designed to masquerade the existence of a piece of code running in hypervisor mode, everybody has become aware that constructing a completely transparent hypervisor is fundamentally infeasible and impractical from a computational and engineering prospective [16]. Indeed, hypervisors introduce several discrepancies, especially in terms of resources and timings. Our goal is to exploit these discrepancies, in

particular timing discrepancies, to detect when the execution environment could not guarantee untampered execution. The main advantage we have over attackers is that checksum validation is performed by an external party, the verifier, that has a real perception of time. We exploit this advantage by including in the checksum function special gadgets that execute instructions that unconditionally trap to the hypervisor. Similarly to exceptions, hypervisor traps cause the CPU to spend several cycles to transition from system (or user) mode to hypervisor mode, to execute the handler of the hypervisor, and to transition back to system mode. By periodically executing such instructions, we cause a noticeable time overhead when a hypervisor is running on the untrusted system.

Figure 5(f) shows a sample gadget we use to detect hypervisors. The gadget reads a word from the memory (line 1), executes a `vmlaunch` instruction (line 3), and then updates the checksum (line 4 and 5). Other instructions, such as `cpuid`, `vmread`, and `vmcall`, can be used for this purpose. The `vmlaunch` instruction is available only on CPUs with hardware support for virtualization. Furthermore, the instruction can be executed only when virtualization support has been enabled. If a hypervisor is running on the untrusted system, any attempt to execute the instruction results in a trap to the hypervisor. In any other situation the CPU refuses to execute the instruction and generates an illegal operation exception. Recall that, by installing a custom IDT, we register handlers for all exception and that these handlers modify the running value of the checksum. In particular, the handler for illegal instruction exception we install additionally updates the address of the faulty instruction for resuming the normal execution of the checksum function from the next one. That is necessary to prevent an endless loop. To do not interfere with the correct checksum computation, after the trap, the attacker has to reproduce the situation that would occur on a system without hypervisor: he has to inject an illegal instruction exception into the guest to trigger the handler registered during the prologue. If the attacker mimics exactly the behavior of the CPU in the absence of the hypervisor, the checksum is computed correctly. However, the cost of the trap, of the execution of the logic to handle the trap, of the event injection, and of the exception handling we have on a system controlled by an attacker operating in hypervisor mode is much higher than the cost of the mere exception handling that we would have on a system without hypervisor. In conclusion, the gadget takes much longer to execute in an insecure execution environment. By executing this type of gadgets multiple times during the checksum loop we have the guarantee that, if the checksum computation produces the correct return value and it does not exceed the expected computation time, the execution environment is tamper-proof.

It is worth noting that if the attacker attempted to execute the checksum function directly in hypervisor mode, he would never be able to regain the control of the execution (this is the same case of an attacker that executes the checksum function in system mode without any hypervisor).

4.3 Obfuscation

After generation, the checksum function is obfuscated using simple obfuscation techniques [17]. Particular efforts are devoted to obfuscate the checksum loop because, by analyzing the loop, the attacker could identify the position of the various gadgets. The strategy we adopt is to introduce specific gadgets for obfuscating the logic of checksum computation. More precisely, these gadgets replace some of the existing gadgets and interrupt handlers with new ones. Furthermore, we obfuscate gadgets singularly by introducing dead code, overlapping instructions, and non-trivial pointers computations.

The gadgets we used for normal checksum computation give, as a side effect, an extra advantage for the verifier over the attacker. The presence of aggressive self-modifying code prevents the attacker from using efficient code emulations techniques, such as dynamic binary translation and software-based virtualization. Indeed, self-modifying code invalidates cached translated code, and forces the emulator to analyze and translate the code again and again. We have experienced directly this problem during the development of Conqueror: self-modifying code executed in system mode caused our development system, based on VirtualBox [18], to trash.

5 Evaluation

5.1 Prototype

We implemented a prototype of Conqueror to evaluate the effectiveness of our proposed solution. The prototype is specific for untrusted 32-bit systems running Microsoft Windows XP, and it consists in a hybrid user/kernel space component, implementing the verifier protocol, and a device driver that stays resident on the untrusted system.

When the verifier wants to bootstrap a tamper-proof execution environment on the untrusted system, it generates a new checksum function and encrypts it. Checksum functions are generated by leveraging a code generation module, currently written in Python. The verifier uses a kernel component to precisely measure packets transmission and arrival times. The kernel component running on the untrusted system passively waits for challenges. When challenged, it fills the TPEB with the encrypted checksum function; when the key is received, the attestation begins. To minimize network latency, both parties intercept challenge requests and responses through a hook installed in the network driver.

To experiment the feasibility of attacks based on hardware-assisted virtualization and their cost we also implemented a minimalistic hypervisor, inspired by the Blue Pill hypervisor [15], that simply resumes normal execution after traps. Obviously, any meaningful hypervisor must be much more sophisticated than this.

5.2 Experimental setup

For our experiments we employed three laptops with the following characteristics: Intel Core2 Duo 2.1GHz, with 4GB RAM, and a Broadcom BCM5906M

network card, connected on the same 100Mbps local network. The first laptop was used as a verifier, the second one as the untrusted system, and the third one as a trusted system. Since our current implementation does not support SMP, on the laptops we used as trusted and untrusted systems we disabled the secondary core of the CPU. In our experiments, the total size of the TPEB and the executable was fixed to six 4Kb pages.

5.3 Estimating the parameters of the challenge

To estimate the various parameters involved in the attestation scheme, we considered two attack scenarios: a dynamic hypervisor-based attack, and a static attack aiming to reverse engineer the checksum function.

To understand how the various parameters of the challenge influenced the overall time to compute the checksum and to understand the opportunities of the attackers, we generated multiple checksum functions, varying the number and type of gadgets and the number of iterations of the checksum loop. After several experiments we decided to fix a minimum for the number of gadgets for "hypervisor detection". In each of the checksum functions we subsequently generated, at least 5% of the total of gadgets performed hypervisor detection.

To estimate the maximum checksum computation time and the network round-trip time (RTT), the verifier relies on a third-party trusted system, with the same hardware characteristics of the untrusted system. It is worth noting that checksum functions can be generated ahead of time and their execution time can be precomputed. Indeed, the running time depends only on the checksum function, on the CPU, and on the amount of data to attest. Given multiple measurements of the checksum computation time, we estimate the maximum computation time using Chebyshev's inequality, that states that for a random variable X, with mean value μ and standard deviation σ, $Pr(\mu-\sigma \leq X \leq \mu+\sigma) \geq 1-\frac{1}{\lambda^2}$, where $\lambda \in \mathbb{R}$. In our context, X is the computation time, including the network RTT[1]. Therefore, the upper bound on checksum computation time is $\Delta_t = \mu + \lambda\sigma$, with confidence $\frac{1}{\lambda^2}$. Similarly, the minimum checksum computation time of the most powerful attacker (i.e., an attacker operating in hypervisor mode) is $\mu - \lambda\sigma$; in the calculation of the minimum computation time of the attacker we assumed the adversary to have a null network overhead.

The number of iterations of the checksum loop must be selected to force the time overhead suffered by the attacker to skyrocket. On the other hand, an excessive number of iterations would increase attacker's opportunities to reverse engineer the checksum function. The challenge is to find the best balance between the two. The approach we used was to generate multiple checksum functions, and to compare the time to compute the checksum in the trusted environment and in the environment controlled by the most powerful attacker. Figure 6 depicts the time overhead suffered by the attacker during our simulations, performed using five different checksum functions. More precisely, the figure shows the difference between the time to compute the checksum on the simulated untrusted

[1] Clearly attestation requires RTT to be minimal. The verifier can measure the RTT and wait to start the challenge if the RTT is too high.

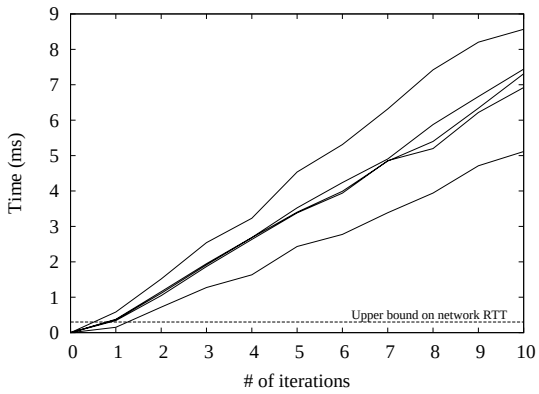

Fig. 6. Time overhead in a hypervisor-based attack

system and on the trusted one. The simulation confirmed our hypothesis: the time overhead suffered by the attacker increases with the number of iterations of the checksum loop. According to our simulation two iterations are sufficient to detect an attack in our experimental scenario (attestation of six memory pages). However, to prevent false negatives, we doubled the number of iterations. Note that the number of iterations to detect a forgery is inversely proportional to the amount of memory to attest; thus, the number of iterations performed by the checksum loop can be tuned accordingly.

5.4 Experimental results

Using the approaches described in the previous paragraphs we generated multiple challenges and used them to verify the effectiveness of Conqueror at detecting authentic checksum computations from forgeries. For clarity we refer to Δ_t, the upper bound of the checksum computation time estimated using Chebyshev's inequality, as the *attacker detection threshold*. In our experiments we chose $\lambda = 11$ to obtain an attacker detection rate with 99% confidence. For each challenge we estimated the attacker detection rate by challenging multiple times the trusted host. Subsequently we challenged the untrusted system twice: once the untrusted host simulated a genuine system (i.e., with no attacker), and once the host simulated the presence of the most powerful dynamic attacker (i.e., an attacker attempting to forge the checksum using a hypervisor-based attack). In all the challenges the untrusted system computed the correct checksum without exceeding the attacker detection rate. Similarly, in all the challenges the untrusted system under the control of the attacker did not compute the correct checksum in time to be considered authentic.

Figure 7 shows the details of one of the challenge we used during the experiment. The figure compares the time the untrusted system took to compute the checksum in the two aforementioned scenarios (the same challenge was repeated more than 50 times). Moreover, the figure shows the attacker detection thresh-

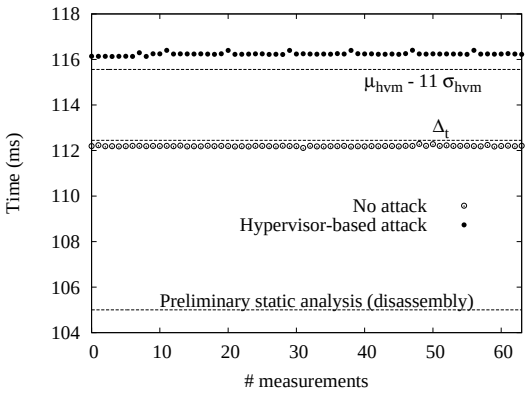

Fig. 7. Checksums computation time in different scenarios

old (Δ_t), and the lower bound for the most powerful attacker ($\mu_{\text{hvm}} - 11\sigma_{\text{hvm}}$). For the challenges in the figure, the average network RTT was less than 0.32ms, and the attacker detection rate was 112.44ms. Similarly, the lower bound for the computation of forged checksum was 115.56ms. The four ms difference and the very small variance between the two clearly indicate that false negatives are practically impossible. The data in the figure confirms the claim: no checksum was forged in time to be considered valid and no authentic checksum was considered forged.

The figure also compares the time requested to compute genuine checksums with the time the attacker would require to perform a preliminary static analysis (i.e., a recursive disassembly) of the checksum function. To measure to cost of the analysis, we loaded in Ida Pro [19], a widely used and well recognized disassembler, the checksum function and then measured the analysis time. Note that the checksum analyzed through Ida Pro was generated without employing any obfuscation technique because the disassembler would not have been able to analyze the code otherwise. The preliminary analysis took about 105ms, just four ms less than the attacker detection rate. Considering that disassembly is fundamental for any static analysis, and that any meaningful analysis to reconstruct the semantic of the checksum function costs much more, it is practically impossible for an attacker to forge a checksum without being detected.

5.5 A real application of Conqueror

Conqueror has been developed to build security applications that must be installed and executed on an untrusted system. All the aforementioned experiments were performed using dummy executables. Nevertheless, to demonstrate the versatility of Conqueror we have developed a special application intended to be run in the tamper-proof execution environment established by Conqueror. The application was a loader for a hypervisor. The goal was use this loader to install a *measured hypervisor* on an untrusted system [1], on-the-fly, and to seg-

regate the untrusted system in a guest virtual machine. We successfully installed the hypervisor on our test untrusted system and then resumed the normal, but controlled, execution of the system. In conclusion, Conqueror represents a pure software alternative to the `senter` and `skinit` operations available in the Intel LaGrande [1] and AMD Pacifica [8] technologies for hypervisors secure *late launch*.

6 Discussion

Conqueror conservatively assumes that if a hypervisor is installed on the system, the hypervisor is malicious. It would be worthless to use Conqueror in a system that runs as a guest of a benign hypervisor: the dynamic root of trust could be established directly by the hypervisor.

The major limitation of Conqueror is the impossibility to bootstrap a tamper-proof environment on SMP and SMT systems. Most modern systems support symmetric flow of executions. An attacker could use the secondary computational resources to forge checksums or to regain control of the execution after attestation. Although we have not addressed the problem in detail, we would like to sketch a possible solution. The verifier can challenge the untrusted SMP (or SMT) system with multiple challenges simultaneously. More precisely, each processor is given a different checksum function to execute. To solve the challenge, the untrusted system has to compute all the checksums and send them back to the verifier, within the given time frame. Thus, the attacker is left with no spare computational resource to use.

7 Related work

The majority of the research work on software-based attestation and verifiable code execution is specific for embedded devices and sensor networks. Most of the schemes are based on the same type of challenge and response protocol [3–6]; they have been thoroughly presented in Section 2. The strength and weaknesses of these schemes have been studied by Castelluccia *et al.* [20]. The approach used in Conqueror is instead inspired by the work done Shaneck *et al.* and by Garay *et al.* [2, 7]. However, the two attestation schemes are also specific for embedded devices and not suited at all for attestation on legacy systems, the target of our work. Genuinity and Pioneer are two schemes, for environment attestation and verifiable code execution respectively, specific for legacy systems [6, 21]. Unfortunately, both schemes are vulnerable to attacks. The vulnerabilities of the former have been studied by Shankar *et al.* [22]. The vulnerabilities of the latter have been introduced in Section 2.

The alternative approach to software-based attestation is hardware-based attestation. The research community spent a lot of efforts in developing hardware technology equipped with special trusted components to make hardware-based attestation practical. Examples of hardware technology with such capabilities are Cerium [23], BIND [24], Intel LaGrande Technology [1], and AMD Pacifica

Technology [8]. In particular, thanks to the efforts of the Trusted Computing Group and the standardization of the TPM chip [25], Intel LaGrande and AMD Pacifica technologies are slowly becoming mainstream. They have been used as ground to develop various hardware-based attestation schemes. Examples of these schemes are the IBM Integrity measurements Architecture [26], the Open Source Loader [27], Terra [28], and Flicker [29]. Similarly to Conqueror and Pioneer, Flicker's goal is to achieve tamper-proof execution of code on untrusted systems. However, while Conqueror and Pioneer are entirely software-based solutions, Flicker leverages the TPM, available on modern commodity hardware, to accomplish the same goal. In particular Flicker relies on a feature introduced in the CPU that allows the secure late launch of virtual machine monitors.

8 Conclusions

We presented Conqueror, a software-based code attestation scheme for tamper-proof code execution on untrusted legacy systems. Conqueror allows to execute an arbitrary piece of code with the guarantee that it is run untampered, even when no specific hardware for trusted computing is available. We developed an experimental prototype of Conqueror, to evaluate its resilience against hypervisor-based attacks, the most powerful type of dynamic attack, and against attacks based on static analysis of the code. By leveraging Conqueror, we also developed a proof-of-concept pure software-based primitive to launch securely a hypervisor in a running untrusted system.

References

1. Grawrock, D.: Dynamics of a Trusted Platform: A Building Block Approach. Intel Press (2009)
2. Garay, J.A., Huelsbergen, L.: Software integrity protection using timed executable agents. In: Proceedings of the 2006 ACM Symposium on Information, computer and communications security (ASIACCS). (2006)
3. Seshadri, A., Perrig, A., van Doorn, L., Khosla, P.: Swatt: Software-based attestation for embedded devices. In: Proceedings of the IEEE Symposium on Security and Privacy. (2004)
4. Seshadri, A., Luk, M., Perrig, A., van Doorn, L., Khosla, P.: Scuba: Secure code update by attestation in sensor networks. In: Proceedings of the ACM Workshop on Wireless Security (WiSe). (2006)
5. Seshadri, A., Luk, M., Perrig, A.: SAKE: Software attestation for key establishment in sensor networks. In: Proceedings of the 2008 International Conference on Distributed Computing in Sensor Systems (DCOSS). (2008)
6. Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., Khosla, P.: Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In: Proceedings of ACM Symposium on Operating Systems Principles (SOSP). (2005) `http://www.cs.cmu.edu/~arvinds/pioneer.html`.
7. Shaneck, M., Mahadevan, K., Kher, V., Kim, Y.: Remote software-based attestation for wireless sensors. In: Security and Privacy in Ad-hoc and Sensor Networks. (2005)

8. AMD, Inc.: AMD Virtualization http://www.amd.com/virtualization.
9. Wurster, G., van Oorschot, P.C., Somayaji, A.: A Generic Attack on Checksumming-Based Software Tamper Resistance. In: Proceedings of the 2005 IEEE Symposium on Security and Privacy. (2005)
10. Intel, Inc.: Intel Virtualization Technology http://www.intel.com/technology/virtualization/.
11. Klimov, A., Shamir, A.: A New Class of Invertible Mappings. In: Proceedings of the 4th International Workshop on Cryptographic Hardware and Embedded Systems. (2003)
12. Robin, J.S., Irvine, C.E.: Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine monitor. In: Proceedings of the 9th USENIX Security Symposium. (2000)
13. Giffin, J., Christodorescu, M., Kruger, L.: Strengthening software self-checksumming via self-modifying code. In: Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC). (2005)
14. Dai Zovi, D.: Hardware Virtualization Based Rootkits. Black Hat USA (2006) http://blackhat.com/presentations/bh-usa-06/BH-US-06-Zovi.pdf.
15. Rutkowska, J.: Subverting Vista Kernel For Fun And Profit. Black Hat USA http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf.
16. Garfinkel, T., Adams, K., Warfield, A., Franklin, J.: Compatibility is Not Transparency: VMM Detection Myths and Realities. In: Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS-XI). (2007)
17. Linn, C., Debray, S.: Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In: Proceedings of the 10th ACM conference on Computer and communications security (CCS). (2003)
18. Sun Microsystems, Inc.: Sun xVM VirtualBox http://www.virtualbox.org/.
19. Hex-Rays: IDA Pro http://www.hex-rays.com/idapro/.
20. C. Castelluccia, A. Francillon, D.P., Soriente, C.: On the Difficulty of Software-Based Attestation of Embedded Devices. In: Proceedings of the 16th ACM conference on Computer and Communications Security (CCS). (2009)
21. Kennell, R., Jamieson, L.H.: Establishing the genuinity of remote computer systems. In: Proceedings of the 12th USENIX Security Symposium. (2003)
22. Shankar, U., Chew, M., Tygar, J.: Side effects are not sufficient to authenticate software. In: Proceedings of the 13th USENIX Security Symposium. (2004)
23. Chen, B., Morris, R.: Certifying Program Execution with Secure Processors. In: Proceedings of the 9th conference on Hot Topics in Operating Systems. (2003)
24. Shi, E., Perrig, A., Van Doorn, L.: BIND: A Fine-Grained Attestation Service for Secure Distributed Systems. In: Proceedings of the 2005 IEEE Symposium on Security and Privacy. (2005)
25. Trusted Computing Group: http://www.trustedcomputinggroup.org/.
26. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and Implementation of a TCG-based Integrity Measurement Architecture. In: Proceedings of the 13th USENIX Security Symposium. (2004)
27. Kauer, B.: OSLO: Improving the Security of Trusted Computing. In: Proceedings of 16th USENIX Security Symposium. (2007)
28. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: a Virtual Machine-based Platform for Trusted Computing. In: Proceedings of the nineteenth ACM symposium on Operating systems principles. (2003)
29. McCune, J.M., Parno, B., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: An execution infrastructure for tcb minimization. In: Proceedings of the ACM European Conference in Computer Systems (EuroSys). (2008)