On the effects of an emulated Memory Trojan on the secure operation of a firewall

Samuel Chenoweth, Sarath Indrakanti and Paul Buckland
Defence Science and Technology Group, Edinburgh, SA - 5111

1 Abstract

The use of commodity computer hardware components has become ubiquitous. They are used in general purpose computers, peripherals such as photocopiers and scanners, personal devices such as cameras and phones, and networking devices such as routers, switches and firewalls. Correct operation of all of these devices depends on the correct operation of the underlying commodity hardware. This hardware is very complex, is designed by large teams of people and is manufactured in remote factories, and often transported long distances before installation. It is possible that deliberate and malicious modifications to commodity computing hardware (i.e. a hardware Trojan) could be made during this process which impact the function of systems implemented using that hardware. In this paper, in the interest of understanding potential advanced threats, we present a concept for a type of hardware Trojan that affects volatile memory operations, i.e. a Memory Trojan. Further, we use an emulated version of the memory Trojan to show how it is able to affect the secure operation of a system by compromising a firewall. We adopt an attacker’s viewpoint in this paper when presenting our memory Trojan concept, the purpose being to help educate the information security community about an emerging class of threat and inform the development of defences and countermeasures.

2 Introduction

Hardware Trojans are a deliberate modification of computer hardware that changes its behaviour, so that its operation is different from what would be expected based on the specifications available to the end user.

The characteristics of hardware Trojans can vary quite widely: there are multiple methods of insertion and activation, a variety of possible locations in which they can be inserted, and there are multiple possible effects. This makes detection of hardware Trojans difficult. A Taxonomy for hardware Trojans is proposed in [1]. See Figure 1.

![Hardware Trojan Taxonomy](image)

*Figure 1: Hardware Trojan Taxonomy, adopted from [1]*
In this paper, we examine the effects of a Hardware Trojan which is able to view and modify volatile memory operations on the memory bus, i.e. a Memory Trojan. As part of this work, we tested our concept of a memory Trojan in an emulation environment. To achieve this, we employed a fully emulated machine running a lightweight operating system, with our memory Trojan code inserted into the code of the emulator to hook volatile memory read and write operations. Volatile memory is included in all general computing hardware and an enormous number of other devices. Any of these computers and devices is a potential target for a Memory Trojan. As an example that demonstrates a dramatic impact on system security, we configured our emulated Memory Trojan to weaken security controls provided by a firewall running on the emulated hardware. The same Memory Trojan could be configured to perform targeted attacks on many other devices and applications.

In enterprise computer networks, a Demilitarised Zone (DMZ) utilising firewalls is typically used to establish a barrier between a secure internal network and an untrusted external network. Firewall hardware may be general purpose computing hardware, or a specialised hardware appliance which is constructed from commodity computing components such as a CPU, flash memory and RAM. This means that either type could be vulnerable to a Memory Trojan, and this has the potential to allow attacks on any vulnerable infrastructure and services within the enterprise network, that would otherwise have been blocked by the firewall.

In Section 3, we discuss related work. Section 4 discusses Memory Trojans in more detail, introducing several types of Memory Trojan. Section 5 discusses the design and operation of the emulated Memory Trojan that we developed. Section 6 outlines some of the issues encountered. Section 7 discusses the implementation using a hardware emulator. Section 8 details an example attack that uses the emulated memory Trojan. Section 9 discusses some possibilities for future work and we conclude the paper in Section 10.

3 Related Work

Several hardware Trojans have been built recently with varying functionality and performance [2-6]. In general, hardware Trojans are hard to detect [7-14] or prevent [15-22], although various countermeasures exist [23]. Much of the work related to memory Trojans is in the area of protecting against memory bus attacks [24, 25], rather than postulating methods of attack.

A limited amount of research has been done on developing hardware Trojans that target the CPU’s use of volatile memory, although in many cases they are better classified as CPU Trojans than as memory Trojans. For example, King et al. [26] designed and implemented two such hardware Trojans, for the purpose of demonstrating the threat posed by hardware Trojans that attack memory. The first of their Trojans consists of extra circuitry in the memory management unit that disables the checking of the current CPU privilege level when handling memory access attempts. A malicious user with unprivileged access to the system can then exploit this, for example by running an unprivileged process that illegally monitors the memory that the operating system uses for receiving keyboard input, in order to sniff for the login credentials of other users. The Trojan in the memory management unit is activated only after circuitry added to the CPU’s memory cache detects a pre-defined trigger sequence in the data that it handles; this allows the Trojan to stay dormant until needed, reducing the chance of its being detected.

The second of King et al.’s [26] Trojans is much more complex and some of its details are unclear, however it appears to be based on malicious firmware code injected into the instruction lines of the CPU cache. The initial injection of malicious firmware into the victim’s system occurs at boot time, which they claim to be able to do using a Trojan in the hardware that initiates the boot operation. When this firmware detects a pre-defined trigger sequence in the cache data, delivered in a UDP packet sent by the attacker (which must be loaded into the cache in order to be inspected by the victim’s machine), additional malicious code that is included with the trigger sequence in the UDP packet is then loaded into instruction cache lines and/or data cache lines reserved by the hardware Trojan, allowing flexible code and data injection that compromises the operation of the process that is executing. This Trojan offers considerable scope and flexibility for the attacker, however successful exploitation would depend on a detailed knowledge (at the machine code level) of the system process that is being targeted. In addition, the need for the injection of malicious firmware at boot time makes this approach rather complex.

Ortega and Muñiz [27] claim to have implemented a hardware Trojan which is inserted in the bus connecting the CPU to the memory. Very few details of their memory Trojan concept are provided in their publications on the
subject, so it is very difficult to assess the extent of the similarities between their work and the memory Trojan explained in this paper. It appears that the approach of Ortega and Muñiz is at least superficially similar to the approach taken in this paper, in that both use a state machine acting as a man-in-the-middle between the CPU and the memory that has the capability to spy on and modify the data being transferred. Like the hardware Trojan explained in this paper, the Trojan described by Ortega and Muñiz also appears to have some ability to be instructed using a command from an external source. However, they have little to say about the details of how these instructions are sent and received. Moreover, they have nothing to say about the way in which the capability could be used to mount practical attacks, which may be non-trivial in many cases. In addition, there is no discussion of the technical issues that must be overcome in order to use the memory Trojan effectively, in particular caching, burst mode (when targeting DMA transfers) and memory re-mapping. This paper addresses these gaps in the literature by outlining a conceptual design for a Trojan in the memory bus, including details of the system for instructing it. Further, the technical issues that are involved are discussed at length. In addition, an emulated implementation of the core features of the design is presented and used to demonstrate a practical attack on a firewall.

4 Memory Trojan Types

A variety of different types of memory Trojan are possible. The most relevant of these are described in this section, although the list is not claimed to be exhaustive.

4.1 Memory Protection Trojans

A memory Trojan could be developed which modifies the behaviour of the memory management unit, so that memory protection mechanisms are weakened. (An example of such a memory Trojan was developed by King et al. [26], as described previously.) This could allow an unprivileged process to read and write to memory that it should not have access to. Note however that the memory management unit is often included in the CPU package, and as such this might be better classified as a processor Trojan. This approach is not explored further in this document.

4.2 Data or Address Bus Modifying Memory Trojans

A data or address bus modifying memory Trojan is a type of hardware Trojan that operates as a man-in-the-middle between the CPU and the memory, so that the memory Trojan may spy on or modify read or write operations that are handled by it. The basic architecture is as shown in Figure 2. Note that the CPU, Memory Management Unit (MMU) and Memory Controller are integrated in many modern systems.

![Figure 2: Bus modifying Memory Trojan architecture](image)

The Memory Trojan may physically reside in the memory controller hardware, on the motherboard (which includes the bus) or in the memory hardware. Technically it could reside in the processor or MMU also, but then it would be better classified as a processor Trojan. If the memory Trojan were to be located in hardware that is integrated with the CPU (i.e. the CPU itself, the MMU and on some systems the Memory Controller), the difficulty of implementing the memory Trojan would be greater than for a memory Trojan in the parts of the architecture that are separate from the CPU.
There are a number of possible approaches to developing a memory Trojan of this class, some of which are outlined below. Of course, it is possible that a Trojan could perform a combination of these, or perform attacks that also involve manipulation of the control bus.

4.2.1 Modification of the data bus for write operations

In this approach, attacks are carried out by modifying the values being written to memory, i.e. changing the data bus when write operations are being handled. This is the simplest approach to use, at least from the point of view of using an existing memory Trojan to mount a practical attack, since a single change written to memory will affect all subsequent read operations from that location, until it is later overwritten. A memory Trojan of this type can then have long lasting effects, even if it is only triggered once. On the other hand, there are greater technical difficulties associated with developing a memory Trojan that modifies the data bus during write operations, since the timing constraints are more severe than for read operations. This issue is not relevant to the emulated memory Trojan discussed in this paper, however.

4.2.2 Modification of the data bus for read operations

In this approach, attacks are carried out by modifying the values returned in response to a read request, i.e. changing the data bus when read operations are being handled. This is a feasible alternative approach to the first approach; however it suffers from the complication that the Trojan would need to modify subsequent read requests from the same location in a consistent manner, if equivalent behaviour is to be obtained.

4.2.3 Modification of the address bus for read operations and/or write operations

In this approach the address bus is changed (possibly as well as modifying the data bus). This causes a read and/or write operation to be applied to the wrong location in memory. For example, the address to which a string is written could be shifted, causing a buffer write overflow or underflow in bug-free code. Worse, data that was supposed to be read from or written to memory owned by an unprivileged process could instead be read from or written to memory owned by the kernel. Although powerful, this type of memory Trojan is more difficult to deploy for many types of attacks (including the one in this paper), in comparison to using a data bus modifying Trojan, since address modification attacks require more knowledge of the layout of data in physical memory.

The remainder of this document is focused on memory Trojans that operate by modifying the data bus only.

5 Design and operation of a Bus Modifying Memory Trojan

The operation of our concept for a data or address modifying memory Trojan proceeds in four phases. These are shown in Figure 3, and explained in the following subsections.
A memory Trojan could be hardwired to perform the same function at all times, for example watching for the byte sequence “AUTO_ADMINISTRATOR_LOGIN=0” on the data bus during write operations and replacing the last byte with “1” when the sequence is detected. This would have the effect of causing a misconfiguration to be loaded into memory when a configuration file containing “AUTO_ADMINISTRATOR_LOGIN=0” is read from disk. However, the static nature of the functionality would be very limiting from an attacker’s point of view, considering that the memory Trojan must be designed and inserted into the hardware well in advance (months if not years) of when the attack is to occur, during which time the details of the desired substitution may need to change.

In order to avoid the limitations of entirely hardwired Trojan functionality, it is in an attacker’s interest to provide some means for modifying the behaviour of the Trojan in the field. The ideal amount of flexibility depends on the attacker’s purpose and strategy, but we expect that at least some details of the operation may need to be configurable. To continue the above example, the memory Trojan could be instructed to search for a different byte sequence and apply a different replacement pattern. It is also desirable to be able to turn the memory Trojan on or off, so that it is only operational when it is needed and/or when it is less likely to cause unintended side effects that increase the risk of detection.

It is not trivial to design the means by which an attacker can supply the instruction message to a hardware Trojan. For a memory Trojan, the options are limited to an out-of-band side channel (e.g. a wireless link connected to the

---

**Figure 3: Flowchart for data bus modifying memory Trojan operation**
Trojan) or use of the buses that the memory Trojan is connected to in any case. The former option is much more likely to lead to detection and is impractical to make use of in environments to which the attacker does not have physical access. The latter approach, in contrast, is stealthy and is often amenable to use by an attacker with limited access and influence.

A variety of methods are possible for encoding an instruction message into the bus traffic seen by the memory Trojan. For example, special patterns of address utilisation and timing could be used. However, the simplest method is to encode the message into the sequence of values that pass through the data bus, since this is something that an external attacker will often have the ability to influence indirectly. When the memory Trojan reads such a sequence, it is important for it to ensure that bytes of the sequence are consecutive, both in time and according to their associated memory addresses; the latter can easily be confirmed by examination of the address bus.

It should be noted that the channel used for instructing a memory Trojan is likely to be unreliable, especially considering the technical issues that are discussed in Section 6. Therefore it is advisable to send a new set of instructions multiple times over, in order to increase the chance that it will be received successfully at least once. In addition, the memory Trojan should be designed to tolerate the receipt of incomplete or invalid instructions.

5.1 Idle Phase

The memory Trojan starts in the idle phase following its initial deployment. Note however that it would need to avoid reverting to the idle phase following a reboot, in order to preserve all state information that existed prior to shutdown. This is a necessary feature for many practical attacks, such as corrupting the copy of a boot time configuration file loaded into memory, as the instructions for doing so may need to be delivered to the memory Trojan in the target machine using services that are themselves started during the boot process (i.e. possibly after the targeted configuration file has been parsed).

In the idle state, the memory Trojan must monitor the data bus to look for a special pre-defined sequence of data values being written to or read from consecutive memory addresses. The purpose of this sequence is to allow an external attacker to send commands to the memory Trojan, whilst distinguishing such messages from normal data. In this paper, this is termed the instruction message identifier. The actual instructions to the memory Trojan immediately follow the identifier, with their associated memory addresses continuing consecutively from the addresses associated with the identifier.

The instruction message identifier should be a sequence of values that are very unlikely to occur by chance in the normal data passing through the data bus. In addition, the sequence must consist of data in a form that can be delivered by an external attacker. For example, a memory Trojan implanted in a mail server could be instructed by including the identifier and instructions in the subject line of an email sent to it by an attacker. In this case, the identifier (and instructions) would be limited to characters that are valid in this context. When the email is processed by the server, at some point the email’s subject line will be written to memory and will thus become visible to the memory Trojan as a sequence of values on the data bus.

When the instruction message identifier is detected, the memory Trojan must transition to the instruction phase.

5.2 Instruction phase

When the memory Trojan sees an instruction message identifier supplied by the attacker’s command-and-control system, the memory Trojan switches to instruction mode to obtain the instruction details that it expects will follow. These details inform the memory Trojan as to the substitution that is to be performed, as well as switching it on or off. Subsequent instruction messages could modify these details and the Trojan’s state any number of times.

After an instruction signal has been identified, the subsequent data received by the memory controller over the data bus needs to be decoded to reveal the instruction details. These instructions provide the memory Trojan with a trigger sequence to look out for among future read / write operations and outline the modifications to the address and
data lines that should be undertaken after triggering. The instructions should also include the ability to set the on/off status of the Trojan, as well as any other configuration settings that may be required, such as whether to interfere with read or write operations.

5.3 Trigger phase

Once the memory Trojan has obtained some instructions that include switching it on, it may begin the next phase. This involves monitoring the data bus on read and/or write operations, seeking the trigger sequence that was provided during the instruction phase. For this memory Trojan design, the bytes on the data bus need to be consecutive in time and by address, in order for them to count as a continuous sequence and therefore relevant for matching to the expected trigger sequence. If the trigger sequence is detected, the memory Trojan transitions to the exploitation phase.

5.4 Exploitation phase

In the exploitation phase, the memory Trojan modifies the data bus and/or address bus during subsequent read and/or write operations, as dictated by the instructions that were received in the instruction phase. In our design, this involves performing substitutions on some of the data bus bytes that follow immediately after the trigger sequence. “Immediately after” must be understood with respect to both time and memory address, so that for an $n$ byte substitution that is to follow the trigger, only the first $n$ bytes passing through the data bus following the detection of the trigger sequence are substituted and only if these bytes are associated with sequential memory addresses in the $n$ bytes following the end of the trigger sequence. If the first $n$ bytes on the data bus following the trigger signal do not have the expected addresses, then the exploitation phase is to be abandoned and the memory Trojan is to return to the trigger phase. Note that while the consecutive address requirement is a feature of our memory Trojan design, we believe that such a feature is necessary for similar memory Trojans to be practical: without it, there could be many incorrect substitutions performed, for example if the exploitation phase is underway when the current process is preempted, so that the remainder of the substitution is performed on unrelated data elsewhere in memory.

Sometimes the data bus bytes that are to be overwritten have values that can be anticipated in advance. In this situation, it is possible to configure a memory Trojan to examine the original values of the data bus bytes during the exploitation phase, in order to confirm that these values are as expected before they are changed. If an unexpected value is detected, the exploitation phase should be abandoned without any further substitution being performed. By using exploitation phase confirmation where it is possible to do so, an attacker deploying such a memory Trojan could significantly reduce the risk of performing substitutions in places that were not intended, thereby reducing the risk of detection.

When the substitution defined in the instructions is complete, the memory Trojan is to return to the trigger phase. In the event that the exploitation phase is abandoned early, the memory Trojan is also to return to the trigger phase. Eventually, another trigger sequence may be detected, with the Trojan then returning to the exploitation phase to perform the corresponding substitution.

We have observed that it is sometimes necessary for the memory Trojan to be triggered multiple times in order to perform the substitution that actually achieves the effect intended by the attacker. One reason for this is the possibility of exploitation phase abandonment, especially if the trigger sequence is not specific enough to eliminate all false positives. Moreover, it is sometimes the case that several complete replacements are required in order to achieve the desired effect, or that the replacement that actually has the desired effect is not the first complete replacement that occurs. An example of this is when a process assembles the original copy of the data of interest (i.e. the trigger sequence and data to be replaced) out of order and/or with other memory accesses interleaved. As the memory Trojan needs to work with sequential data in order to trigger, this original copy of the data will not be modified by the memory Trojan. If the process then makes multiple copies of this unmodified data elsewhere in memory for various purposes, with the data handled sequentially, then the memory Trojan is able to be triggered and to perform the replacement on these copies, and the desired effect could depend on any combination of the substitutions being performed successfully. The need for multiple replacements could also arise in other situations,
for example if the memory Trojan was idle when the original copy of the data of interest was first written to memory (for a memory Trojan modifying data during write operations) or if it was idle when the original copy in memory was first read to create other copies (for a memory Trojan modifying data during read operations). It is also possible that data stored on disk is loaded into memory multiple times and that the desired effect requires the replacement to be applied to the data each time, or that a copy of the data of interest was in the cache when the first replacement occurred and so a second replacement must be applied to this data when it is later restored to memory.

Note that during all phases, the hardware Trojan should also look out for a new instruction identifier. If a new instruction identifier is received, it must immediately return the memory Trojan to the instruction phase to override the previous instructions. However, if incomplete or invalid instructions are obtained then the memory Trojan should resume the phase that it was in, using the most recent instructions it has that are both complete and valid.

The effect of the substitutions performed during the exploitation phase is at the discretion of the attacker sending the instruction signals. A typical effect is to disable or weaken some key security mechanism through corrupting the configuration data loaded into memory from a file, but there may be many other possibilities. More is said on this matter in Section 8.

6 Issues with Bus Modifying Memory Trojans

Bus modifying memory Trojans have a number of technical issues that may affect their successful deployment by an attacker, some of which are explained in this section. This section also considers some of the special circumstances in which these difficulties are reduced or eliminated, and some techniques that an attacker could use to mitigate these difficulties. Where they apply, the issues raised in this section work in favour of the defence; it is even possible that some of them could be used by computer hardware engineers or operating system developers to increase the difficulty of a memory Trojan based attack, for example by avoiding consecutive mapping between virtual and physical memory pages (see Section 6.5).

6.1 Caching

It is important to be aware of the effect of caching on architectures such as this. Typically, there are two levels of caching for memory operations, which means that in many cases the memory Trojan will not have access to all read / write operations. This has the potential to prevent the memory Trojan from working reliably in situations where the memory operations are served by the cache rather than by main memory. However, memory Trojans based on data bus modification for memory write operations will be unaffected by caching when attacking DMA transfers, since DMA typically bypasses all caches. Note that all disk read operations involve a DMA transfer that writes to memory. For this reason, caching issues are not expected to prevent attacks that are based on corrupting the loading of configuration files, such as the attack outlined in Section 8.

6.2 Sequential Memory Access Reordering

A sequence of memory accesses issued by software that is running can often be reordered by the memory controller without affecting the operation of the software. For example, if the sum of the entries of an integer array is accumulated in a CPU register using a loop over an addition instruction, it does not matter what order the memory read operations occur in. Many memory controllers take advantage of situations where the order does not matter, by reordering the memory operations to improve overall performance. Note however that even where the order of memory operations is not important to the software that is running, it is usually of critical importance for a data bus modifying memory Trojan, since this typically depends on the trigger sequence bytes (and the subsequent bytes that are to be replaced) being encountered in the expected order. If the memory controller were to reorder the stream of bytes that are relevant to the memory Trojan, then it is likely that the Trojan would fail to be triggered or fail to complete the substitution.

Data bus modifying memory Trojans are still practical in many cases, despite the memory access reordering issue. There are a number of reasons for this. Firstly, when memory accesses are reordered, this usually occurs in data units larger than one byte: 16 bytes is a typical size. The memory controller may change the order in which these
blocks of data are handled, however order is usually preserved within each block. In some cases, the data of concern to the memory Trojan is contained in a single block; in these cases the Trojan operation would not be affected by reordering performed by the memory controller. Secondly, when the memory controller handles a long series of memory operations to sequential addresses, the middle of the series is often not reordered, since there is little performance advantage to be gained by reordering memory accesses that are already in sequence. Thirdly, reordering does not occur very often for DMA transfers, since these often involve the transfer of large blocks of data in sequence, for which reordering offers little opportunity for performance gain.

6.3 Need for Attacking DMA Transfers

When the memory Trojan is used to compromise the copy of a configuration file written to memory from disk, it is in the attacker’s interest if the replacement occurs during the DMA transfer for the low level disk read operation. Some reasons for this are avoiding cache issues and avoiding memory access reordering, as noted previously. In addition, attacking the DMA transfer reduces the need for the attacker to understand the details of how the data will be processed. During the DMA transfer all the data for an entire block of the configuration file is written to memory in sequence, with searches and replacements by a memory Trojan being possible anywhere within this block. For example, it may be possible for the search target string and replacement to span multiple lines of the configuration file, even if the data from the block transfer is later parsed a line at a time. In contrast, if the memory Trojan performed its search and substitution only when a copy of a configuration file fragment in memory is later being parsed a line at a time using non-DMA memory accesses, then the memory write operations for sequential lines of the file are likely to be separated by other memory operations, thereby breaking the continuity that the memory Trojan requires for its operation. In order to avoid all these issues, DMA transfers are often the most effective target for a data bus modifying memory Trojan to attack, and so the memory Trojan should be designed to support attacks on them.

6.4 Burst Mode

On typical hardware platforms, memory accesses frequently occur in burst mode. This is a type of bulk transfer operation that reduces the overheads associated with the alternative, namely performing the transfer as a long series of individual memory operations. In burst mode, for example, it is possible to specify the address of the start of a block write operation on the address bus, followed by a long series of values on the data bus that are written to subsequent addresses of the block. Similar efficiencies can also be obtained by eliminating repeated signals on the control bus. If the system being attacked implements some type of burst mode but the memory Trojan is not designed to support this, then many potential attacks based on the memory Trojan could be thwarted. Note that DMA transfers usually occur in burst mode, so an inability to attack burst mode transfers would be particularly limiting for the attacker, given the previous discussion on the value of attacks on DMA transfers. Therefore, it is in an attacker’s interests to design the memory Trojan to support attacks on burst mode transfers.

6.5 Physical versus Virtual Addresses

The addresses available to the memory Trojan in this architecture are physical addresses, as opposed to the virtual addresses seen by a process running on the CPU. This has both advantages and disadvantages for an attacker using a memory Trojan.

The primary advantage for an attacker is related to the fact that the translation between virtual addresses and physical addresses occurs in the memory management unit (MMU), which is also involved with memory protection through the page fault mechanism. If the memory Trojan were to use virtual addresses, it would need to be located in the bus connecting the MMU to the CPU, since this is the side of the MMU where virtual addresses are used. In this case, the memory Trojan would need to interfere with the page table updating process (i.e. in addition to manipulating the address and/or data buses), or else its capabilities would be limited by the memory protection system. More specifically, a memory Trojan between the CPU and MMU that does not interfere with page table updates would not be able to use address modification to access memory that the currently running process is forbidden from accessing by the operating system. Adding hardware to interfere with page table updates would greatly complicate the engineering required to develop the memory Trojan. By using a memory Trojan that works
with physical addresses on the bus on the memory side of the MMU, all memory protection mechanisms are bypassed without the need for interfering with page table updates; this allows unfettered access to memory.

The primary disadvantage for an attacker is that address translation can cause memory that was contiguous in virtual address space to become non-contiguous in physical address space. The memory Trojan will not generally be able to predict how this will occur, which limits its ability to compensate for it. If a page boundary occurs in the middle of one of the data byte sequences that the memory Trojan needs to detect or modify, this could prevent the memory Trojan from working correctly. This is illustrated in the diagrams below, which show some typical mappings between virtual address space and physical address space, and the effect that these could have on the ability of a memory Trojan to work with an area of memory (denoted in the diagrams by “TARGET STRING”) that it expects to be contiguous. Note that for the memory Trojan described in this document, the contiguous area of memory would need to include both the trigger sequence and the area to which the replacement is to be applied that follows it. Contiguous memory is also required for providing the instruction sequence that configures the memory Trojan.

In the example below, the target string is entirely contained within a single virtual memory page, and therefore also contained within a single physical memory page. It is likely that the target string will be contained within a single memory page if the target string is very short in comparison to the memory page size (typically 4096 bytes), however this becomes increasingly less likely as the length of the target string approaches the page size and impossible once the target string is longer than a page. In the case of the target string being entirely contained within a memory page (as shown below), the operation of the memory Trojan is unaffected by the remapping, regardless of its complexity.

**Figure 4: Scenario where the target string is entirely contained within one memory page**

In the example below, the target string is split over a virtual memory page boundary. However, the remapping function happens to preserve the continuity of the pages in physical memory, allowing the memory Trojan to operate as normal. Such a simple remapping function is sometimes used, if sufficient memory is available and memory fragmentation is low.

**Figure 5: Scenario where the target string spans memory pages that are consecutive in physical memory**

In the example below, the target string is split over a virtual memory page boundary. In addition, the remapping function does not preserve the continuity of the pages in physical memory. This means that the memory Trojan is unable to operate on the target string, unless special techniques are used to compensate for the remapping (at least partially).
We have investigated a technique that might provide at least a partial ability to handle matching across page splits with discontinuous mapping. The idea behind this technique is that discontinuities caused by page splits can only occur at the end of a memory page, never in the middle of a page. Therefore, when the memory Trojan is processing memory operations that are sequential in time and confirming that the addresses are also sequential, it can rely on page splits being absent as long as the next address is in the same physical memory page as the last address, and therefore perform the address check as normal. However, when the next address is in a different physical memory page then a page split could have occurred, and so the address check can be relaxed to allow any address that differs from the expected address by an integer multiple of the page size. Subsequent addresses would then need to be sequential from that point, until the end of that physical page is reached. For example, in the situation shown in Figure 6, the “S” of the “TARGET STRING” would be expected to be at physical address 28672, in the absence of a page split. If the next memory access in time is for an “S” at physical address 12288 instead, then the match can continue because 28672 and 12288 differ by an integer multiple of the page size. The “T” that is expected to come next would then need to be at physical address 12289 rather than 28673 for the match to continue, because there cannot be a discontinuity within a page. Clearly there are risks associated with relaxing the address check in this way, because there is a chance of continuing a match onto a page that is from an entirely unrelated virtual address space, for example a page owned by a different process. The effectiveness of this technique has not been confirmed through rigorous testing with the emulator. Moreover, it is possible that there could be practical difficulties with using this technique in real hardware. At this stage we cannot say more on this matter.

7  Implementation with Bochs

A demonstration of the type of memory Trojan discussed in the document has been developed using Bochs (running within Ubuntu), which is an open source system for full emulation of PC hardware and firmware. A memory Trojan was simulated by adding hooks in the source code locations relating to the emulation of memory read or write operations. The simulated memory Trojan code that is invoked from these hooks implements the concepts and algorithms outlined in this document for the triggering and exploitation phases.

The emulated memory Trojan has a few functional differences to the memory Trojan concept thus far described. Firstly, instructions are not provided to the memory Trojan using information encoded into the memory bus; instead, the memory Trojan is instructed from the host operating system by loading a configuration file (config.hwt, in the main Bochs directory) at bootup, as well as whenever the command "touch command.hwt" is executed in the main Bochs directory. Secondly, the emulated memory Trojan does not include any measures for supporting burst mode.

8  Example Attack on a Firewall

In this example, we used the emulated memory Trojan described in Section 7 to compromise the firewall managed by the TTY Linux operating system installed on the emulated machine. This demonstrates the basic functionality of the memory Trojan, as well as showing how memory Trojans could be used to undermine the infrastructure on which enterprises depend for their security. In order to show the risks that such attacks could entail, we then demonstrate an exploit that is facilitated by the compromised firewall.

8.1  Compromising the Firewall with the Emulated Memory Trojan
It is possible to configure the emulated memory Trojan to turn the TTY Linux firewall off altogether. However, that would have very obvious effects that would be likely to be discovered quickly. It is better to use a more subtle attack, such as allowing the firewall to be started as normal but with just one extra port opened. In TTY Linux, the file `/etc/firewall.conf` is used to specify the details of the firewall configuration, including which specific ports are opened to incoming connections. (All outgoing ports are opened by default under TTY Linux, while incoming ports are closed by default unless listed in the relevant place in the configuration file.) The configuration file is shown below:

```plaintext
# /etc/firewall.conf
# This configuration file specifies INPUT ports on which to ACCEPT connections.
# The default crosslinux iptables firewall setting that is not controlled in
# this configuration file:
#
# 1. DROP on FORWARD
# 1. DROP on INPUT
# 2. ACCEPT on OUTPUT
# 3. ACCEPT on INPUT belonging to established connections having the
#    state ESTABLISHED or RELATED
# 4. Un-allowed incoming TCP connection attempts are logged and dropped.
#
# Do not allow packet forwarding. Uncomment the "[forward]" string below to
# disable all FORWARD filtering.
#
# [forward]
#
# Allow ICMP ping requests and responses.
#
# [ping]
#
# Specify the TCP ports on which connections will be allowed.
#
# [tcp]
# ftp   # port 21
# ssh  # port 22
# http # port 80

# Specify the UDP ports on which connections will be allowed.
#
# [udp]
# tftp   # port 69
# 1024:65535 # Allow access to unprivileged ports.
```

If the attacker desires to open an extra inbound TCP port, this can be achieved by using a memory Trojan to modify the copy of this file that is written to memory during parsing. Specifically, the line "http # port 80" can be replaced with "http:81 # Pwnd", thereby opening all TCP ports in the range from port 80 (http) to port 81, i.e. opening just one extra port. Note that the presence of the comments in the configuration file is helpful for the attacker, since it allows extra syntactical information to be inserted without overwriting any existing syntax and without requiring searches or replacements over multiple lines. The memory Trojan configuration required to do this is shown below:

```plaintext
hwt_enabled=true
hwt_type=write
verbose_hwt=false
multi_replace=true
memory_page_size=4096
allow_page_split_matches=false
target_pattern=http # port 80
mask          =MMMMBBBBBBBBBB
replacement   =http:81 # Pwnd
address_shift=0
replace_timeout=0.1
```
The most important items in the above configuration are explained below. See also Figure 7, which shows an example of how this configuration affects the operation of the emulated memory Trojan.

- **hwt_enabled=true**: Turns the emulated memory Trojan on.
- **hwt_type=write**: Attack memory write operations rather than memory read operations.
- **target_pattern=http # port 80**: A sequence of bytes that includes those that the memory Trojan will search for, in order to trigger a replacement. Note that only those bytes identified for matching by the mask are actually used for this purpose.
- **replacement=http:81 # Pwnd**: A sequence of bytes that includes those to be used as the replacement after the memory Trojan is triggered. Note that only those bytes identified for replacement by the mask are actually used for this purpose.
- **mask=MMMMBBBBBBBBBB**: This determines how the target pattern and replacement are used. An M means that the character in the same position of the target pattern must match for the memory Trojan to be triggered. An R means that the corresponding character in the replacement pattern. A B is similar to an R, except that the character about to be replaced is checked against the corresponding character in the target pattern, with the memory Trojan aborting the replacement and returning to the trigger phase if these do not match. A space can also be used in the mask, which indicates that the corresponding byte is not required to match the target pattern and is not to be replaced.

### Table: Stream of Pre-existing Bytes, Target Pattern, Mask, Replacement

<table>
<thead>
<tr>
<th>Phase: Trigger phase</th>
<th>Exploitation phase</th>
<th>Trigger phase</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Stream of Pre-existing Bytes:</th>
<th>p o r t</th>
<th>2 2 \n h t t p #</th>
<th>p o r t</th>
<th>8 0 \n \n #</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target pattern:</td>
<td>h t t p #</td>
<td>p o r t</td>
<td>8 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mask:</td>
<td>M M M M</td>
<td>B B B B B B B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replacement:</td>
<td>h t t p : 8 1</td>
<td>#</td>
<td>P w n d</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7: A possible sequence for the triggering and exploitation of the emulated memory Trojan, based on the configuration used for the attack on the TTY Linux firewall. The top row shows an example stream of bytes handled by the memory Trojan (before they are modified by it), with the earliest arriving byte on the left and the latest arriving byte on the right. The position of each byte in the row also indicates the address associated with it, so that the second byte in the row has an address that is one greater than the first byte in the row, etc. This diagram also shows how the target pattern, mask and replacement match up with the byte streams for a successful triggering and completed replacement. The bottom row shows the stream of bytes after being modified by the replacement. Note that this diagram is equally applicable to memory Trojans that target read operations or those that target write operations.

In order for the memory Trojan to have an effect on the deployed firewall, the above memory Trojan configuration must be active while TTY Linux is booting. (Alternatively, the commands "service firewall stop" followed by "service firewall start" can be used to cause the modified configuration to take effect without requiring a reboot.)

The changes to the firewall configuration that are caused by this attack can be seen using the Linux “iptables –L” command. On a system with the default firewall configuration (and no active memory Trojan), the output of this command includes the following lines, which comprise a list of the TCP ports that the firewall will allow new inbound connections on. Without the memory Trojan operating, new inbound connections are only allowed for ftp (port 21), ssh (port 22) and http (port 80):

```
ACCEPT tcp -- anywhere anywhere tcp dpt:ftp ctstate NEW
ACCEPT tcp -- anywhere anywhere tcp dpt:ssh ctstate NEW
ACCEPT tcp -- anywhere anywhere tcp dpt:http ctstate NEW
```
With a memory Trojan configured to compromise the firewall by opening the additional TCP port 81, the output of the “iptables –L” command instead includes the following lines:

```
  ACCEPT  tcp  --  anywhere  anywhere  tcp dpt:ftp ctstate NEW
  ACCEPT  tcp  --  anywhere  anywhere  tcp dpt:ssh ctstate NEW
  ACCEPT  tcp  --  anywhere  anywhere  tcp dpts:http:hosts2-ns ctstate NEW
```

It can be seen that the memory Trojan has caused an additional TCP port to be opened, here identified as “hosts2-ns”, which is port 81.

### 8.2 Exploiting the Compromised Firewall

With TCP port 81 opened (using either of the above memory Trojan attacks), it is now possible for an attacker operating from a different machine on the network to connect to any service that happens to be listening on this port. This would not be possible if the firewall had kept this port closed. If the service listening on the port has any vulnerabilities, then these could then be exploited by the attacker.

Suppose for example that there was a logging service (/root/sillyservice/sillyservice) listening on port 81, which consists of the following shell script:

```bash
#!/bin/bash
set +e
while true; do
    nc -l -p 81 | ./vulnerable
done
```

The above shell script invokes an executable (vulnerable) compiled from the following C source code (vulnerable.c):

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_LINE_LENGTH 1000
int main(int argc, char *argv[])
{
    char line[MAX_LINE_LENGTH+1];
    char command[MAX_LINE_LENGTH+1];
    while (!feof(stdin))
    {
        line[0]="\0";
        fgets(line, MAX_LINE_LENGTH, stdin);
        if (line[strlen(line)-1]==\n') line[strlen(line)-1]=\0';
        if (strlen(line)>0)
        {
            sprintf(command, "echo $s\n" >> log.txt", line);
            fprintf(stderr, "Executing: $s"
", command);
            system(command);
        }
    }
    return 0;
}
```

The above service takes any input it receives from port 81 and writes it to the file log.txt. However, it does this by constructing an echo command that incorporates the user input, which is subsequently executed as a system call. This is vulnerable to command injection, if a malicious user supplies input that includes the double quote character used to terminate the string.
For example, an attacker can connect to sillyservice from a machine elsewhere on the network, using a command such as "ncat 131.185.228.147 81", where the IP address in the command is that of the emulated TTY Linux machine. The attacker can then type a series of commands to sillyservice, each terminated using the Enter key. The basic formula for injecting commands is as shown below:

"; injected_command #

When a line similar to the above is provided as input, the string constructed by vulnerable is as shown below:

echo ""; injected_command "" >> log.txt

When the system call is made, the string is interpreted as a blank echo command followed by injected_command, with a comment at the end that is ignored by the shell. This allows the attacker to execute arbitrary system commands on the emulated TTY Linux machine, at the same privilege level as sillyservice.

In order to view any output from an injected command, it is necessary for an attacker to start a listener on his/her local machine, using a command such as "ncat -l 12345". While the listener is running, the attacker can then provide input to the remote sillyservice as described above (which involves a separate ncat instance acting as a client). To obtain feedback from an injected command, the basic formula to use is as shown below:

"; injected_command | nc 131.185.228.11 12345 #

Note that the IP address in the above must be the IP address of the attacker's machine, while the port number must match the port number used by the listener.

9 Related and Future Work

Hardware Trojans, and especially memory Trojans, may become a significant threat to ICT security in future years. State sponsored entities such as foreign intelligence agencies have the capability and motivation to launch sophisticated cyber-attacks against Australian public and private ICT systems, potentially including attacks based on hardware Trojans. Such attacks may even be within the capability of other hostile actors such as organised crime groups. Although the difficulty and cost of developing and deploying such attacks may be high in many cases, the rewards to adversaries of a successful attack could be substantial, including the theft of national defence secrets, theft of valuable research and other intellectual property, sabotage of critical infrastructure, etc. As countries such as Australia do not have effective control on the hardware manufacturing process and supply chain, it is very difficult for them to mitigate this risk by attempting to keep its ICT hardware free of hardware Trojans. Detection of hardware Trojans is also notoriously difficult, due to the enormous variety of possibilities for developing hardware Trojans and their inherent subtlety. For these reasons, we believe it is necessary to assume that hardware Trojans will be present in future ICT hardware, and that ICT systems will therefore need to be resilient enough to tolerate their presence. Therefore, we have explored the remediation of hardware Trojan threats, through the development of architectures that mitigate the most serious security risks that hardware Trojans introduce. There is considerable scope for future research in this area.

10 Conclusion

Hardware Trojans deliberately modify computer hardware in order to change its behaviour, so that its operation is different from what would be expected based on the specifications available to the end user. In this paper, our focus was on memory Trojans, which are a type of hardware Trojan. We described a conceptual design for a bus-modifying memory Trojan that has the necessary flexibility for use in practical attacks. We then discussed some issues in designing and implementing this type of memory Trojan. In order to demonstrate the practicality of the theory, we implemented an emulated version of the core memory Trojan design by modifying the code of a hardware emulator. The utility of this implementation for conducting practical attacks was then shown, through its use in compromising the firewall included in the TTY Linux operating system installed on an emulated machine.
This paper helps to demonstrate the seriousness of the hardware Trojan threat, as well as the need for effective countermeasures.

11 References


