The Grenade Timer: Fortifying the Watchdog Timer Against Malicious Mobile Code*

Frank Stajano † ‡ Ross Anderson †

† University of Cambridge Computer Laboratory,
New Museums Site, Cambridge CB2 3QG, United Kingdom
Tel: +44 1223 334600; Fax: +44 1223 334678
Email: *name.surname*@cl.cam.ac.uk

‡ AT&T Laboratories Cambridge,
24a Trumpington Street, Cambridge CB2 1QA, United Kingdom
Tel: +44 1223 343000; Fax: +44 1223 313542
Email: fms@att.com

Abstract

Systems accepting mobile code need protection from denial of service attacks staged by the guest program. While protected mode is the most general solution, it is not available to the very low-cost microcontrollers that are common in embedded systems.

In this paper we introduce the *grenade timer*, an evolution of the watchdog timer that can place a hard upper bound on the amount of processor time that guest code may consume. Unlike its predecessor, it is resistant to malicious attacks from the software it controls; but its structure remains extremely simple and maps to very frugal hardware resources.

Keywords: mobile agents, security, denial of service, watchdog timer.

1 Introduction

We move towards a scenario of ubiquitous computing: all around us, payment cards, vehicles, consumer electronics, white goods and office equipment already have computing capabilities, which will be further enhanced when coupled with short range wireless telecommunications facilities [2, 9, 12]. In the future, we may expect computing and networking facilities to become embedded in all the objects that surround us, from clothes to doorknobs.

Mobile code is now an established development in distributed systems, and is already part of the deployed Internet infrastructure in so far as most desktop browsers run Java. Cellular telephones have already become miniature web browsers and it will not take long for vendors to want to push new functionality into their customers' phones or even SIM cards, much like web sites nowadays do with Java applets and ActiveX controls. More advanced forms of mobile code have also been envisaged, such as itinerant agents [4].

It seems plausible to expect that ubiquitous computing and mobile code will eventually intersect, with small pieces of code roaming around from one appliance to another. There will be many uses for such an arrangement, of which automatic firmware updates, configuration management, resource discovery and personalisation will probably be seen as the least imaginative. However useful the application, though, mobile code will ultimately be unacceptable to consumers without proper system-level safeguards, otherwise its introduction will dramatically decrease reliability and, ultimately, personal safety. Interesting though it may be to have a fridge capable of reordering food over the Internet when it finds itself empty, the conventional fridge may still be preferable if the wired one is vulnerable to viruses that randomly defrost it or turn it off, spoiling its contents in the process[1].

To protect the host system from untrusted mobile code, one should first explicitly identify the security properties to be safeguarded. Farmer *et al.* [8] offer an excellent taxonomy of the security issues for mobile agents, but place the emphasis on the more difficult problems of protecting the agent, compared to which the protection of the host is easy. So does Ordille [13], who examines trust as the agent roams along a path not determined in advance. Much of the literature on security tends to concentrate on the more the-

*Appears in "Proceedings of 7th International Workshop on Mobile Multimedia Communications (MoMuC 2000)", Waseda, Tokyo, Japan, 23–26 October 2000.

[1]Let us not forget that viruses were the earliest and arguably still the most widespread instances of mobile code.

oretically challenging problems: plenty of attention is devoted to *confidentiality* (keeping information secret), some to *authenticity* (verifying the identities of the principals involved) and *integrity* (preventing unauthorised modifications), but very little to *availability* (ensuring that the system keeps on working). In our case, though, like in many other real-world scenarios including banking and emergency services, the priorities are reversed, and the most pressing concern is to avoid denial of service attacks. We shall therefore concentrate on those. In particular, we wish to be able to limit the amount of processor time that the guest program will consume.

This is especially important for devices that integrate several functions. Our practical experience with frauds against digital prepaid electricity meters [1] suggests the following insight. If you imagine a multifunction meter containing not only the system software from the electricity company but also some code from the user's bank which takes care of reloading the meter with money when necessary, you will see that the electricity company needs to be sure that the banking program cannot take over control of the processor. Otherwise, if a poorly (or maliciously) written banking application got stuck in an endless loop, the meter would no longer be able to turn off the power once the available credit balance were exhausted.

2 The watchdog timer

The most general mechanism for limiting the resources consumed by mobile code is probably a processor with a protected mode facility, on which the operating system can implement pre-emptive multitasking. The guest program is run in real mode and, when its allotted time expires, it is stopped by the operating system running in protected mode. Address protection also prevents the guest program from reading or writing memory outside its allotted segment, thereby addressing first-order confidentiality and integrity concerns[2].

Where the hardware does not provide a protected mode facility, running mobile code safely is more difficult. Emulating protected mode in software is not usually a viable option, since checking the legitimacy of every instruction being run leads to unacceptable performance. Static verification before execution, as in Java, requires that the language in which the program is expressed be suitably restricted. This is not unreasonable for mobile code, if only for portability reasons, but it means that the host must run a virtual machine instead of native code. The overhead and performance penalty are negligible on a desktop machine, but might be significant for the tiny microprocessors used in ubiquitous computing devices. Accepting only object code that was previously checked and signed by some trusted authority, as happens in the ActiveX architecture, may be appropriate in some circumstances but gives fewer guarantees on the actual behaviour of the code, while at the same time imposing more serious centralised restrictions and/or vulnerabilities.

Practically all modern microprocessors offer a protected mode, so the discussion about alternatives may at first appear to have only historical interest. However we should not underestimate the fact that, at the scale of deployment envisaged for the ubiquitous computing scenario, one of the primary operational directives will be "low cost", and that a sub-dollar difference in the cost of the processor may make the difference between success and failure in the marketplace. Besides, embedded applications tend to use single chip microcontrollers, for which the availability of a protected mode (with the associated complications in the memory management architecture) is still much less common than for microprocessors for desktop systems.

Another observation is that a server based on the latest microprocessor and pre-emptive multitasking OS is such a complex system that people are now forced to look with suspicion at claims of availability guarantees based on the properties of its kernel. It is not uncommon for builders of mission critical systems that must run 24×7 to adopt a "belt and braces" attitude and supplement those kernel guarantees with a hardware *watchdog timer*.

The concept of a watchdog timer is powerful yet deceptively simple. The timer is basically a counter that is decremented at every clock tick and that will reset the processor when it reaches zero. It is used to ensure that a software crash will not hang an unattended machine. The software must periodically prove to be working by reloading the counter; if it fails to do that for too long, the watchdog will eventually reboot the system.

The watchdog timer was originally developed for high reliability embedded systems such as those used for industrial process control. The rationale was that EMI and electrical surges in a noisy environment could flip bits of the memory at random and therefore cause the execution of meaningless code, which had the potential to lock up the machine indefinitely. Commercial implementations are available either as an external chip or as integrated in the microcontroller [5, 6, 7]. But watchdog timers are now also used in PC-based servers; they are available as PC expansion cards [3, 10] (sometimes they are even included on the motherboard) and are supported by several BIOS manufacturers [11, 14].

[2]Second-order problems are still possible, such as violations of confidentiality through the exploitation of covert channels, but we won't go into details here.

3 The grenade timer

The watchdog timer, however, despite its usefulness for controlling a program that might accidentally run astray, cannot protect the system against a malicious program that purposefully tries to keep the processor to itself. The operating system could tell the guest program "I am about to execute you, and I will grant you up to 20 million clock cycles; if you have not returned control to me by then, you will be terminated". But if by hypothesis we use a processor where all code runs in real mode, the above is pointless: without the distinction between real mode and protected mode, if the operating system can load the counter with the value "20 million cycles", then so can the guest program, which is therefore free to extend its own lifetime indefinitely.

We were thinking about this problem in the context of providing a restricted execution facility for mobile code that might run on the hardware node developed at AT&T Laboratories Cambridge for the Prototype Embedded Network (previously known as Piconet) project [2]; this portable short-range wireless system uses the TMP93PS40 16-bit microcontroller from Toshiba (based on the TLCS-900/L processor) which, like most other microcontrollers, does not offer a protected mode.

Inspired by the watchdog timer concept, we came up with a construction that solves the above problem. We call it the *grenade timer*. It allows a system without protected mode to limit the execution time of a guest program.

The novel idea is to build a counter that can not be reloaded once it is running. An egg-timer set to count downwards for five minutes can easily be stopped or wound back at any time during its countdown; however a hand grenade whose fuse has been lit will definitely explode after the set number of seconds, regardless of any attempt to put the pin back in. We want our counter to be like a grenade: once the operating system has pulled out the pin and tossed the fizzing grenade to the guest process, no authority in the world (not even the operating system itself) must be able to postpone the scheduled explosion.

We may however provide an escape mechanism that makes the explosion happen immediately: this is equivalent to firing a bullet into the fizzing grenade to make it blow up prematurely. This is a useful thing to do when a well-behaved guest program wants to return control to the operating system before having consumed its entire quota of allocated cycles.

We complete the metaphor by explicitly saying that the operating system wears a thick armour that protects it from explosions: when the grenade goes off (which corresponds to the processor been reset), the guest process dies, but the operating system is unharmed.

This simple construction adds a "poor man's protected mode" to the processor: as long as the grenade is fizzing, the system is in real mode and the executing process can't escape the fact that it will have to relinquish control when the grenade goes off[3]. When the grenade is not fizzing, the running process is by definition in protected mode, because it has the power to set the delay and pull the pin to start the countdown if and when it wishes.

It may be instructive to compare the relationship between the watchdog timer and the grenade timer with that between the CRC (cyclic redundancy check) and the cryptographically strong one-way hash.

Both the CRC and the hash are functions that take as input a bit string of arbitrary length and return as output a short bit string of fixed length. In both cases, the purpose of the output is to act as error-detecting redundancy: if the input string is changed, the precomputed redundancy will no longer match, and a verifier will be able to detect the change. The difference between the two is that the CRC only resists to random errors, while the hash (which is more expensive to compute) also resists to malicious attacks. The CRC is designed to detect random bit errors such as those due to telecommunication noise; but an attacker may flip unimportant bits in a forged message until it yields the desired CRC, therefore making the forgery undetectable. The hash, however, is designed to stop just that, and makes it computationally infeasible to generate a message with a given hash output.

The situation is somewhat similar with our timers. The watchdog timer is designed to interrupt programs that might accidentally lock up, but is insufficient to stop programs that actively try to circumvent it. The grenade timer, instead, explicitly addresses the case of malicious attacks. Informally,

$$\text{watchdog} : \text{grenade} = \text{CRC} : \text{hash}.$$

At the circuit diagram level (see figure 1), the external interface of the grenade coprocessor as described so far (which contains little more than a counter) is as follows:

The `RESET` input brings the device in a known initial state: it is a necessary system-level facility outside the control of the main processor and we shall not consider it any further here.

The device[4] is in real mode when `IS_FIZZING` is 1 and in protected mode otherwise. The `LOAD` input and the `COUNT_VALUE` data bus may be used to load

[3]The position to which the processor jumps on reset will always run in protected mode and therefore needs to be safeguarded from unauthorised changes, otherwise the guest program could substitute itself to the operating system code and bypass any protection. See section 4.

[4]Or, more precisely, the system that incorporates it.

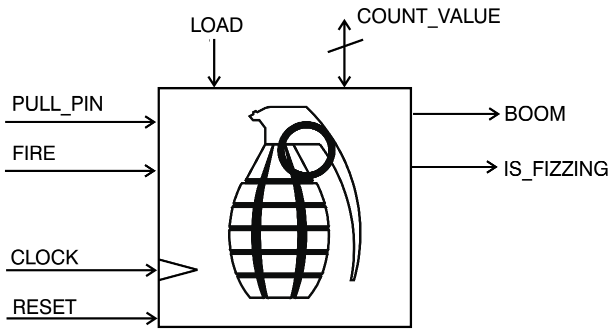

Figure 1: External interface of the grenade timer.

a new counter value, but only in protected mode, i.e. when IS_FIZZING = 0.

The PULL_PIN input only works from protected mode. When raised, it causes the device to start counting down from the currently loaded count value; IS_FIZZING goes to 1 as the count starts.

When the counter reaches 0, the BOOM output (to be connected to the main processor's RESET line) goes to 1 for one clock cycle and IS_FIZZING comes back to 0.

The FIRE input only works[5] while IS_FIZZING is 1; it forces the explosion to happen (BOOM will go to 1 for one cycle) and the fizzing to stop, but it does not reset the counter to 0, so that the operating system can read the current COUNT_VALUE to see how many ticks remained when the grenade exploded.

4 Limiting the addressable range

The mechanism so far described only protects agains the guest program not relinquishing control, but not against integrity or confidentiality violations. This is serious: if the guest can overwrite the OS code, any protection can be circumvented. Placing the OS in ROM, as is frequently done in embedded systems, makes it safe from modifications; but the working area would still be subject to corruption. Besides, in an environment supporting dynamic updates to the system, the OS could be held in Flash or RAM.

This issue may be addressed by augmenting the grenade timer with some extra lines that intercept the processor's address bus (see figure 2). The grenade timer, while fizzing, masks out the top bits of the address bus (even only one bit is sufficient in principle if one wishes to save on pin count) and replaces them with a fixed page address defined by the operating system before pulling the pin. This prevents the guest program from reading or writing memory or I/O addresses outside the authorised page.

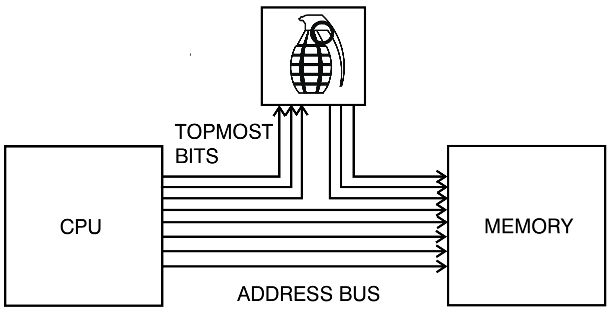

Figure 2: Restricting the range of accessible addresses.

There are however some subtleties to do with passing control from the OS to the guest and with calling OS services from the guest. We have thought of mechanisms to deal with those issues, which we shall briefly sketch below, but we don't find them particularly elegant, so suggestions and criticisms from readers will be welcomed.

To call the OS from the guest, the program pokes a system call request into a well-known memory location designated as a mailbox, protects it with a CRC for integrity, and raises FIRE. This resets the processor, which jumps into the OS. The start-up code of the OS checks the mailbox and, if the CRC matches, proceeds with the evaluation of the request (other criteria will decide whether to honour the request, including sanity checks on the parameters and the amount of ticks left over in the grenade timer when it was shot). The CRC is then deleted by the OS, so that the presence of a valid one can be taken as a guarantee of freshness; this allows the OS to distinguish a system call from a genuine reset due to external causes. Finally, the grenade timer is restarted and the guest code is reentered, as described below.

To pass control from OS to guest we envisage a special section of code spanning the page boundary. The code starts in protected mode, raises PULL_PIN as it crosses the boundary and then finds itself in real mode with the grenade fizzing. The obvious limitation of the fixed entry point may be overcome by using another mailbox in a similar way.

5 Conclusions

We introduced the *grenade timer* as an inexpensive mechanism for imposing a hard limit of the CPU time that a guest program may consume, in the absence of a protected mode for the host processor.

[5]Since FIRE is unresponsive when PULL_PIN works and vice versa, an attempt to minimise pin count might combine PULL_PIN and FIRE into a single input line whose function would depend on the state of IS_FIZZING. Here, for clarity, we prefer to keep the two signals distinct.

Today the majority of embedded systems are still based on simple microcontrollers that do not offer a protected mode. If we extrapolate and merge the scenarios of ubiquitous computing and of mobile agents, we obtain a world in which even embedded devices will customarily run mobile code. In those circumstances, a true protected mode architecture seems the most appropriate way to support safe sandboxing.

In the interim, though, for as long as the hardware designer finds the best trade-offs in a simpler microcontroller that does not offer that feature, the grenade timer may prove useful. It does not require many more gates than a simple counter and will therefore easily fit in a corner of the general purpose FPGA that the system almost certainly already includes.

6 Acknowledgements

We are grateful to Paul Osborn of AT&T for useful comments on an earlier draft.

References

[1] Ross J. Anderson and S. Johann Bezuidenhoudt. "On the Reliability of Electronic Payment Systems". *IEEE Transactions on Software Engineering*, **22**(5):294–301, May 1996. http://www.cl.cam.ac.uk/ftp/users/rja14/meters.ps.gz.

[2] Frazer Bennett, David Clarke, Joseph B. Evans, Andy Hopper, Alan Jones and David Leask. "Piconet: Embedded Mobile Networking". *IEEE Personal Communications*, **4**(5):8–15, Oct 1997. ftp://ftp.uk.research.att.com/pub/docs/att/tr.97.9.pdf.

[3] Berkshire Products. "ISA PC Watchdog Board User's Manual". Manual, Berkshire Products, Suwanee, GA, 2000. http://www.berkprod.com/docs/isa-wdog.pdf.

[4] Davis Chess, Benjamin Grosof, Colin Harrison, David Levine, Colin Parris and Gene Tsudik. "Itinerant Agents for Mobile Computing". *IEEE Personal Communications*, **2**(5):34–49, Oct 1995. http://www.research.ibm.com/massdist/rc20010.ps.

[5] Dallas Semiconductor. "Using the High-Speed Micro's Watchdog Timer". Application Note 80, Dallas Semiconductor, Oct 1999. http://www.dalsemi.com/datasheets/pdfs/app80.pdf.

[6] Dallas Semiconductor. "Using the Secure Microcontroller Watchdog Timer". Application Note 101, Dallas Semiconductor, Nov 1999. http://www.dalsemi.com/datasheets/pdfs/app101.pdf.

[7] Dallas Semiconductor. "Watchdog Timekeeper". Application Note 66, Dallas Semiconductor, Jul 1999. http://www.dalsemi.com/datasheets/pdfs/app66.pdf.

[8] William M. Farmer, Joshua D. Guttman and Vipin Swarup. "Security for Mobile Agents: Issues and Requirements". In "Proceedings of the 19th National Information Systems Security Conference", pp. 591–597. Baltimore, Md., Oct 1996. http://csrc.nist.gov/nissc/1996/papers/NISSC96/paper033/SWARUP96.PDF.

[9] Jaap Haartsen, Mahmoud Naghshineh, Jon Inouye, Olaf J. Joeressen and Warren Allen. "Bluetooth: Visions, Goals, and Architecture". *ACM Mobile Computing and Communications Review*, **2**(4):38–45, Oct 1998.

[10] ICS Advent. "Model PCI-WDT 500/501 Product Manual". Manual 00650-144-1A, ICS Advent, San Diego, CA, 1998. http://www.icsadvent.com/techlib/manuals/00650144.pdf.

[11] ITOX, Inc. http://www.itox.com/pages/products/LitTig/TCub/bulletintc1a.htm.

[12] Kevin J. Negus, John Waters, Jean Tourrilhes, Chris Romans, Jim Lansford and Stephen Hui. "HomeRF and SWAP: Wireless Networking for the Connected Home". *ACM Mobile Computing and Communications Review*, **2**(4):28–37, Oct 1998.

[13] Joann J. Ordille. "When agents roam, who can you trust?" In "First Conference on Emerging Technologies and Applications in Communications (etaCOM)", Portland, OR, May 1996. http://cm.bell-labs.com/cm/cs/doc/96/5-09.ps.gz.

[14] Phoenix Technologies. http://www.phoenix.com/platform/awardbios.html.