Using Memory Errors to Attack a Virtual Machine

Sudhakar Govindavajhala
Department of Computer Science
Princeton University
sudhakar@cs.princeton.edu

Andrew W. Appel
Department of Computer Science
Princeton University
appel@cs.princeton.edu

Abstract

We present an experimental study showing that soft memory errors can lead to serious security vulnerabilities in Java and .NET virtual machines, or in any system that relies on type-checking of untrusted programs as a protection mechanism. Our attack works by sending to the JVM a Java program that is designed so that almost any memory error in its address space will allow it to take control of the JVM. All conventional Java and .NET virtual machines are vulnerable to this attack. The technique of the attack is broadly applicable against other language-based security schemes such as proof-carrying code.

We measured the attack on two commercial Java Virtual Machines: Sun's and IBM's. We show that a single-bit error in the Java program's data space can be exploited to execute arbitrary code with a probability of about 70%, and multiple-bit errors with a lower probability.

Our attack is particularly relevant against smart cards or tamper-resistant computers, where the user has physical access (to the outside of the computer) and can use various means to induce faults; we have successfully used heat. Fortunately, there are some straightforward defenses against this attack.

1 Introduction

Almost any secure computer system needs basic protection mechanisms that isolate trusted components (such as the implementation and enforcement of security policies) from the less trusted components. In many systems, the basic protection mechanism is hardware virtual memory managed by operating system software. In the Java Virtual Machine (and in the similar Microsoft .NET virtual machine), the basic protection mechanism is type checking, done by a *bytecode verifier* when an untrusted program is imported into the system.

Assuming the type system is sound (like Java, but unlike C or C++), type-checking as a protection mechanism allows closer coupling between trusted and untrusted programs: object-oriented shared-memory interfaces can be used, instead of message-passing and remote procedure call across address spaces. Thus, language-based mechanisms are very attractive—if they work.

Because the untrusted programs run in the same address space as trusted parts of the virtual machine, type checking must provide strong protection. The Java Virtual Machine Language type system has been proved sound [8, 9], and subsets of it have even been proved sound with machine-checked proofs [19]. Provided that there are no bugs in the implementation of the verifier and the just-in-time compiler, or provided that one can type-check the output of the just-in-time compiler using an approach such as proof-carrying code [5], type-checking should be able to guarantee—as well as virtual memory can—that untrusted programs cannot read or write the private data of trusted programs.

Java can be compiled to efficient machine code, and supports data abstraction well, because it uses link-time type-checking instead of run-time checking. However, this leaves Java vulnerable to a time-of-check-to-time-of-use attack. All the proofs of soundness are premised on the axiom that the computer faithfully executes its specified instruction set. In the presence of hardware faults, this premise is false. If a cosmic ray comes through the memory and flips a bit, then the program will read back a different word than the one it wrote.

A previous study of the impact of memory errors on security measured the likelihood that a random single-bit error would compromise the security of an existing program [20]. This study found (for example) that a text-segment memory error would compromise `ssh` with

about 0.1% probability. Boneh et al. used random hardware faults to recover secrets in cryptographic protocols [3]. Anderson and Kuhn studied various physical attack techniques on smartcards and other security processors by inducing errors at specific locations at specific instants [1, 2]. Unlike them, we use arbitrary errors to take over a virtual machine.

We show that when the attacker is allowed to provide the program to be executed, he can design a program such that a single-bit error in the process address space gives him a 70% probability of completely taking over the JVM to execute arbitrary code.

An attacker could use this program in two ways. To attack a computer to which he has no physical access, he can convince it to run the program and then wait for a cosmic ray (or other natural source) to induce a memory error. To attack a tamper-resistant processor to which he has physical access only to the outside of the box (such as a Java card), he can induce it to run the program and then induce an error using radiation or other means; we will describe measurements of the effects of infrared radiation.

One might think that parity checking or error-correcting codes would prevent this attack. But in the low-profit-margin PC market, parity or ECC bits are usually not provided.

This paper highlights the importance of hardware reliability in assuring the security of a program.

2 The attack program

Our attack is against a JVM that permits untrusted code to execute after it has used its bytecode verifier to check that the code is type-safe, and therefore respects its interfaces.

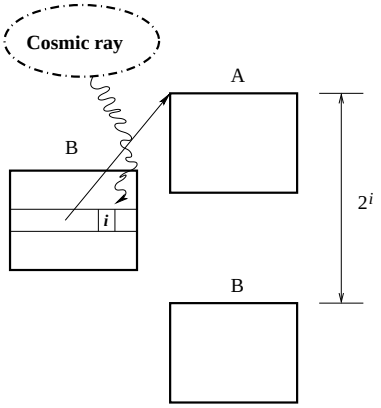
The goal of our attack applet[1] is to obtain two pointers of incompatible types that point to the same location. This permits circumvention of the Java type system. Once the type system is circumvented, it is straightforward to write a function that reads and writes arbitrary memory locations in the program address space, and hence executes arbitrary code [10, pp. 74–76].

The attack works by sending the Java Virtual Machine a program (which the JVM will type-check using the bytecode verifier) and waiting for a memory error. The program type-checks; when it runs, it arranges the memory so that memory errors allow it to defeat the type system.

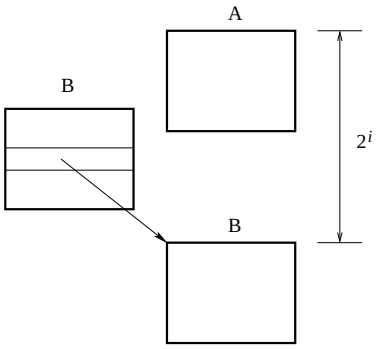
Our attack applet is quite simple. First, it fills the heap with many objects of class B and one object of class A. All the fields of all the B objects are initialized to point to the unique A object, which sits at address x. Classes A and B are defined so that, including the object header, their size is a power of two:

```
class A {          class B {
 A a1;               A a1;
 A a2;               A a2;
 B b;                A a3;
 A a4;               A a4;
 A a5;               A a5;
 int i;              A a6;
 A a7;               A a7;
};                 };
```

Now the applet waits patiently for a memory error. Suppose a cosmic ray flips the ith bit of some word in a B object:

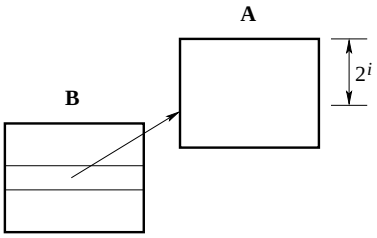


If 2^i is larger than the object size, then $x \oplus 2^i$ is likely to point to the base of a B object ($\oplus$ is the exclusive-or operator):



Thus, there's a field whose static type is A but which actually points to a B object; this can be exploited, as we will explain. On the other hand, suppose 2^i is smaller than the object size; then the mutated field points within the A object:

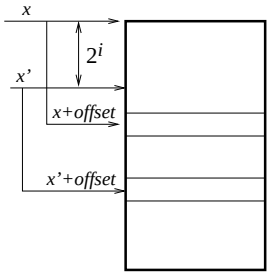
[1] An *applet* is a program that runs with few privileges: no access to the file system and limited access to the network.



Suppose there is a pointer variable *p* of class A, containing address *x*. When the program dereferences the *b* field of *p* into a pointer *s* of type B, as follows:

```
A p;  B s;
s = p.b;
```

it is really fetching from address $x + offset$, where *offset* is the distance from the base of the object to the beginning of the *b* field:



But if the *i*th bit of *p* has flipped, then the fetch is from address $(x \oplus 2^i) + offset$, as shown in the diagram. The applet dereferences `p.b`; it thinks it's fetching a field of type B, but it's really fetching a field of type A.

Now that we explained the principle of how our attack applet works, we will explain some details of the algorithm. Figure 1 summarizes the layout of objects in memory created by the attack applet. There is one object of class A; let us suppose it is at address *x*. The applet sets all the A fields of all the objects to point to *x*, and it sets the field `x.b` to point to some object of class B.

After creating the data structure, it repeatedly reads all the *A* fields of all the objects and checks (via Java pointer equality) whether they still contain *x*.

Now suppose that in one of the many B objects, one of the bits in one of the fields has been traversed by a cosmic ray, and flips — for example, bit 5 of field `a6` of record `b384`. We fetch this field into a pointer variable *r* of type A:

```
A r;  B b384; B q;
r = b384.a6;
q = r.b;
```

The field `b384.a6` originally contained a copy of *p*, as did all the A fields of all the objects. If the *i*th bit of `b384.a6` has been flipped, then when the program dereferences *r.b* it is fetching from address $(x \oplus 2^i) + offset$ into *q*. Most of the program memory is filled with fields of

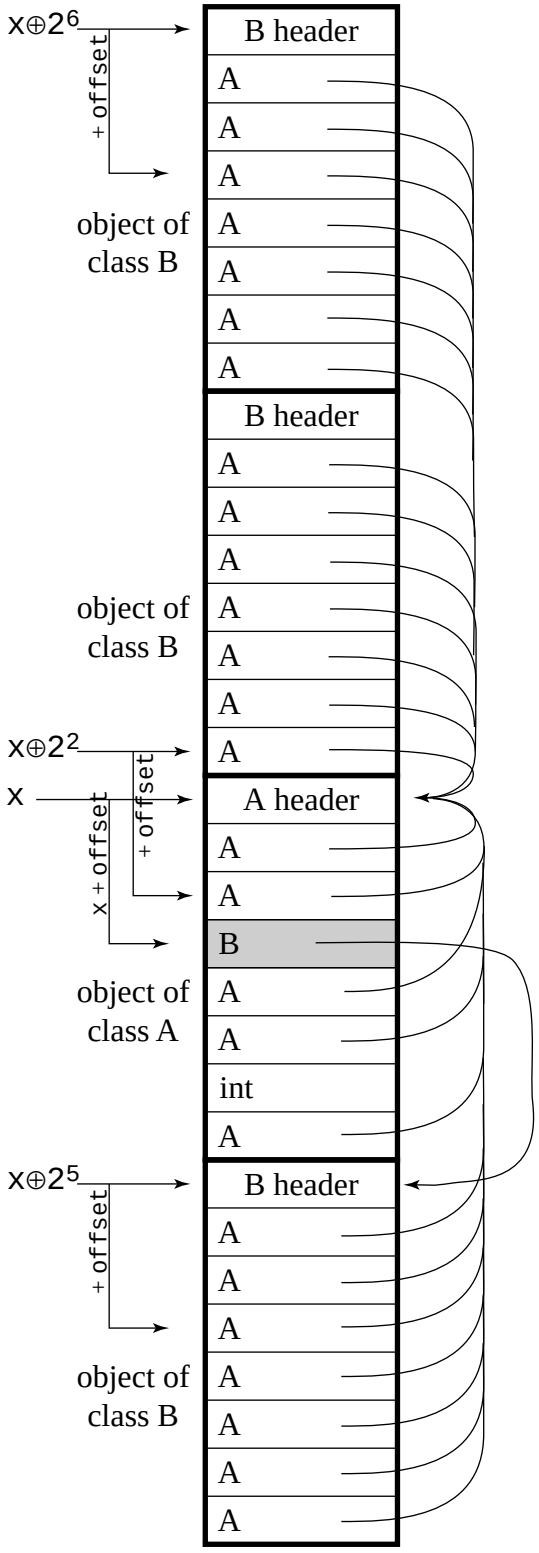


Figure 1. Attacker memory layout

type *A*, each field containing *x* (*x* is the address of the object of type *A*). Hence, the address $(x \oplus 2^i) + \mathit{offset}$ is very likely to contain *x*.

For example, in Figure 1 if bit 2, bit 5, or bit 6 has flipped, then memory location $(x \oplus 2^i) + \mathit{offset}$ contains a pointer of type A.

Now we have a pointer variable *q* whose static type is B but which contains a pointer of type A — a circumvention of the type system. We also have a pointer variable *p containing the same address* whose static type is A. This can be used to write a procedure that writes arbitrary data at arbitrary location.

3 Exploiting a type-system circumvention

Once we have equal pointers *p* and *q* of types A and B, we can take over the virtual machine. Consider the code fragment:

```
A p;
B q;
int offset = 6 * 4;
void write(int address, int value) {
   p.i = address - offset ;
   q.a6.i = value ;
}
```

The value `offset` is the offset of the field `i` from the base of an A object. This procedure type-checks. The fields `i` of type *A* and `a6` of type *B* are at equal offsets from their bases. Suppose that through our attack, *p* and *q* contain the same address. The first statement writes `address - offset` at the field `q.a6`. The second statement writes `value` at an offset of `offset` from `q.a6`. Thus the procedure writes *value* at *offset + (address - offset) = address*.

For any address *a* and value *v*, the call `write`(a,v) will write *v* at address *a*. The method to read arbitrary addresses is similar. This can be exploited to execute arbitrary code by filling an array with machine code and then overwriting a virtual method table with the address of the array. Once the attacker can do this, he can access any resource that the trusted parts of the virtual machine can access.

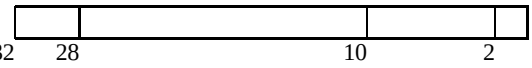
There are simpler (and more portable) ways to achieve security violations than writing and executing machine code. For example, every Java runtime system defines an object of a class called SecurityManager that enforces security policies controlling such things as access to the filesystem by untrusted applets. A pointer to this object is available through the method *System.getSecurityManager*. Normally, the static typechecking of the Java bytecode verifier is effective at preventing classes other than SecurityManager from writing the *allowFileAccess* field of the security manager. But once our exploit has a way to write to arbitrary locations, it's easy to alter any field of the security manager and thus circumvent any policy that the security manager is enforcing.

4 Analysis

We can predict the effectiveness of this attack. Let *M* be the number of bytes of program data space, filled up with one A and many B objects. Let each object contain *s* words (including the object header of *h* words), and let 2^w be the word size. Then the number of objects is $N = M/(s \cdot 2^w)$.

We call two objects "cousins" if their addresses differ in a single bit. Let the "cousin number" of *x* be $C(x)$, the number of objects of type *B* whose address differs from *x* by a single bit. Suppose the object size is a power of two, number *N* of objects is a power of two, and the objects are all contiguous in memory; then it's obvious that $C(x)$ will be $\log_2 N$. If we relax these assumptions, then it's plausible that $C(x)$ might still be approximated by $\log_2 N$. Figure 2 shows the actual values of $C(x)$ for all the objects in a particular run of IBM's commercial JVM, and it shows that $\log_2 N$ is an excellent predictor of $C(x)$.

Suppose the word size is 4, and the object size is 1024, and consider a 32-bit pointer value as follows:



In any of the many A fields in the memory, we can exploit a bit-flip in any of the $C(x)$ bits from bit 10 to bit 27; any one of these flips will cause the pointer to point to the base of one of the B objects.

We can also exploit any flip in bits 2 through 9; this will cause the pointer to point within the object (or within an adjacent object), and when the *offset* is added to the pointer we will fetch one of the many nearby A fields within the A object (or within an adjacent B object). This pointer won't point at an object header, so if we attempt to call a virtual method (or to garbage-collect), the JVM will probably crash; but fetching a data field (instance variable) does not involve the object header.

We cannot exploit flips in the extreme high-order bits (the resulting pointer would point outside the allocated heap) or the two low-order bits (which would create a pointer not aligned to a word boundary).

Let $K = C(x) + \log_2 s$; *K* is the number of single-bit errors we can exploit in a single word, so it is a mea-

Cousin number	# of objects with that cousin number	
0	2	
1	2	
2	13	
3	7	
4	20	
5	59	
6	30	
7	0	
8	0	
9	0	
10	0	
11	0	
12	0	
13	614	
14	2,868	
15	32	
16	29,660	
17	110,640	
18	282,576	
Mean	Total	$\log_2$(Total)
17.56	426,523	18.70

Figure 2. Measured cousin number distribution in the IBM JVM.

sure of the efficiency of our attack. The $C(x)$ component comes from exploitation of high-order bit flips; the $\log_2 s$ comes from exploitation of medium order bit flips (pointers within the object). Our attack is extremely efficient. We were able to obtain a K value of 26 on a 32-bit machine.

For each pointer of type A that contains x, a bit flip in any of K bits would result in a successful exploit. (A bit flip in other bits may result in the pointer pointing to garbage.) If we have N objects containing $N(s-h)$ pointers of type A which contain x, any single-bit flip in $KN(s-h)$ bits can be exploited. We can make Ns almost as large as the process address space, and we can minimize the overhead of object headers by using large objects.

We can estimate the efficiency of the attack as the fraction of single-bit errors that allow the attack to succeed. We assume the following parameters:

P bytes of physical memory on the computer,

M bytes available to the Java program in its garbage-collected heap,

w is the $\log_2$ of the word size,

s is the number of words in an object,

h is the number of words occupied by the header of each object.

Then the number of objects is $N = M/(s \cdot 2^w)$, the number of exploitable pointers is $N(s-h)$, the number of exploitable bits per pointer is $K = \log_2 N + \log_2 s$. Thus the fraction of exploitable bits in the physical memory is

$$\frac{N(s-h)(\log_2(Ns))}{8P}$$

Multiple-bit errors will also allow the attack to succeed. As long as the flipped bits are all contained within the K exploitable bits, then the memory error will allow type-system circumvention, except for the rare case that the corrupted value, when the offset is added to it, ends up pointing to an object header. To minimize the likelihood of pointing to a header if a few bits are flipped, we want the Hamming distance from $x + offset$ to the base of the object to be high; that is *offset* should be an integer with several 1 bits.

Suppose we have M bytes of memory, and some small number d of bits flip. If the flipped bits are all in different words, then this is essentially like several single-bit attacks, provided that none of the bit-flips is in a place that crashes the JVM or the operating system.

Suppose d different bits flip, all in the same word (uniform randomly distributed), with word size W bits. Then the probability that all d are within the K bits that we can exploit is $\frac{K\cdot(K-1)\cdots(K-d+1)}{W\cdot(W-1)\cdots(W-d+1)}$. For $K = 26$ (which is the highest value of K that we have observed), we can still exploit a 6-bit error with about one-fourth the likelihood of a one-bit error.

5 Experimental results

We implemented and measured our attack against two different commercial Java Virtual Machines, both running on RedHat Linux 7.3:

- IBM's Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.1);
- Sun's Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.1_02-b02).

Notwithstanding the coincidence in build numbers, these appear to be quite different virtual machines.

We ran several sets of experiments:

1. We ran a privileged Java thread inside the JVM that uses the Java Native Interface to a C-language function that flips a bit in the process address space. This serves mostly to check the operation of the attack applet and confirm our closed-form analysis.

2. We ran an unmodified JVM, with a separate privileged Linux process that opens /dev/mem and flips a random bit in the computer's physical memory. This simulates a naturally induced memory error that results from a cosmic ray, as described in Section 6.

3. We ran an unmodified JVM, and induced memory errors by heating the memory to 100°C, as described in Section 7.

In order to minimize the proportion of memory devoted to object headers, we used objects of approximately 1024 bytes; our A and B classes had 249 pointer fields and (on the IBM JVM) 3 header words.

IBM's JVM allowed the applet to allocate up to 60% of the physical memory, but not more. The JVM reveals sufficient information about the address of the object to compute the cousin number for each object. We optimized the attack to use this information. We refer the reader to the appendix for details about the optimization.

Software-injected in-process faults:

The JVM permitted a process address space of 467 megabytes on a machine with 1 GB of memory. We were able to allocate 422066 objects. A bit flip in any of the bits $2 \ldots 27$ of any pointer resulted in a successful attack; that is, $K = 26$.

Thus we were able to use $\frac{422066 \cdot 249 \cdot 26}{8 \cdot 467 \cdot 2^{20}} = 0.70$ of the bit flips in the program address space.

Software-injected anywhere-in-physical-memory faults: We were able to allocate $N = 57,753$ objects on a machine with 128 MB RAM. We flipped a random memory bit in the physical memory using the /dev/mem interface. We expect a success probability of $\frac{57753 \cdot 249 \cdot \log_2(57753 \cdot 249)}{8 \cdot 128 \cdot 2^{20}} = 0.32$. We ran 3,032 trials of the experiment. By comparing the pointer fetched from the memory with a pointer to the object, we detected that a bit flipped in 1353 trials. Of these 1353 times, we were able to take over the JVM 998 times (the remainder were in an unexploitable bit of the word, and hence the JVM crashed). In 1679 trials, the bit flip was not detected by our program; of these trials, there were 23 where the operating system crashed, and at most 22 trials where our JVM crashed[2]. Our efficiency was 0.33, which is close to the analytic prediction.

Sun's JVM allowed the applet to allocate up to 60% of the physical memory, but not more.

Software-injected anywhere-in-physical-memory faults: We were able to allocate $N = 61,181$ objects on a machine with 128 MB RAM. We flipped a random memory bit in the physical memory using the /dev/mem interface. We expect a success probability of $\frac{61181 \cdot 249 \cdot \log_2(61181 \cdot 249)}{8 \cdot 128 \cdot 2^{20}} = 0.34$. We ran 292 trials of the experiment. By comparing the pointer fetched from the memory with a pointer to the object, we detected that a bit flipped in 154 trials. Of these 154 times, we were able to take over the JVM 112 times (the remainder were in an unexploitable bit of the word, and hence the JVM crashed). In 138 trials, the bit flip was not detected by our program; of these trials, there were 4 where the operating system crashed. Our efficiency was 0.38, which is close to the analytic prediction.

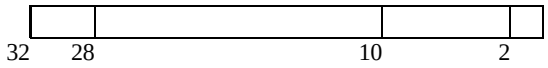
Exploiting before crashing. If errors occur frequently, then the raw efficiency (what fraction of the errors can be exploited) may not be as important as the likelihood of exploiting an error before the JVM or the operating system crashes. If p is the probability that an individual memory error leads to a successful exploit, and q is the probability that an individual memory error crashes the JVM or the operating system, then the probability[3] that the successful exploit occurs before the machine crashes is $p/(p+q)$. Our measurement shows (of the IBM JVM) a value of $p = 0.33$, $q = 0.12$, so $p/(p+q)$ is about 73.3%.

Safe bit flips. In our applet, almost the whole memory is filled with pointers pointing to the single A object. The applet repeatedly tests these pointers against the pointer to the A object to detect a bit flip. If the bit flipped is in the extreme high/low bits, dereferencing the flipped pointer might crash the JVM because the pointer points outside the address space or to an unaligned address. How do we find out if the program can safely dereference the flipped pointer? Suppose the word size is 4, and the object size is 1024, and consider a 32-bit pointer value as

[2] In our logs, there are 22 trials where it is not clear whether the JVM crashed. To be conservative, we assume that the JVM crashed in those cases.

[3] The argument is as follows: With each error, we win with probability p and we play again with probability $(1-p-q)$. Thus the likelihood of eventually winning is $p \cdot \sum_{i=1}^{\infty}(1-p-q)^i$, or $p/(p+q)$.

follows:



If the bits flipped are in the bits $2 \ldots 27$, then dereferencing the flipped pointer is safe. If the flipped bits are in the bits $10 \ldots 27$, the new pointer should point to one of the B objects. Thus, we can detect if the bits flipped are in the bits $10 \ldots 27$ by comparing the flipped pointer with each of the B objects. The program has no safe way to distinguish a flip in the bits $2 \ldots 9$ from a flip in the bits $0 \ldots 1$ and $28 \ldots 31$. Thus, if we have flips in the bits not known to be in $10 \ldots 27$, we have to dereference the pointer and hope it is safe to dereference.

By comparing against the B objects and detecting if the bits flipped are in the bits $10 \ldots 27$, and using only these safe flipped pointers for the attack, though our efficiency is lower, we have a better win-before-lose-ratio. In this case q, the probability that an individual memory error crashes the JVM or the operating system drops to 45/3032 and hence the probability that the successful exploit occurs before a machine crash is $p/(p+q) = 93.6\%$. In this version, where we do not use the flips in the bits $2 \ldots 9$ (corresponding to the interior offset of the fields in the object), the optimal object size for our exploit is smaller. Smaller object size would allow us to use flips in more bits per word, while increasing the object header overhead. Our analysis shows that for a JVM that uses 2 header words per object, the optimal object size is 128, with the win-before-loss ratio for this object size being 94.6%.

6 Susceptibility of DRAM chips

To attack machines without physical access, the attacker can rely on natural memory errors. Memory errors are characterized as hard or soft. Hard errors are caused by defects in the silicon or metalisation of the DRAM package and are generally permanent (or intermittent) once they manifest. Soft errors are typically caused by charged particles or radiation and are transient. A memory location affected by a soft error does not manifest error upon writing new data.

Soft errors have been studied extensively by the avionics and space research communities. They refer to soft errors as "single event upset" (SEU). In the past, soft errors were primarily caused by alpha particles emitted by the impurities in the silicon, or in the plastic packaging material [21]. This failure mode has been mostly eliminated today due to strict quality control of packaging material by DRAM vendors.

Recent generations of DRAM chips have been made more resistant to memory errors by avoiding the use of boron compounds, which can be stimulated by thermal neutrons to produce alpha particles [11]. Currently the probable primary source of soft errors in DRAM is electrical disturbance caused by terrestrial cosmic rays, which are very high-energy subatomic particles originating in outer space.

It is hard to find good recent quantitative data on the susceptibility of DRAM chips to radiation-induced faults. The most informative paper we came across is from IBM, and is for memory technologies several generations old [14]; in 1996 one might have expected one error per month in a typical PC.

Since then, changes in DRAM technology have reduced its radiation-induced fault rate. Dynamic RAMs are implemented with one capacitor to hold the state of each bit of memory. The susceptibility of a DRAM cell to faults is proportional to its size (cross-section to cosmic rays), and inversely proportional to its capacitance. As new capacitor geometries have implemented the same capacitance in less chip area, the fault rate per bit has significantly decreased [16]. Even though these technology changes were not made with the primary intent of reducing the error rate, they cause DRAMs to be much more reliable than a decade ago. It appears that one will have to wait for several months on a desktop machine for an error.

DRAMs are most susceptible when the data is being transferred in and out of the cells. An attack program would do well to (miss the cache and) frequently access the DRAM chips.

In the near future, we may expect errors not just from cosmic rays but from the extremely high clock speeds used on memory busses [15]. The faults will not occur in the bits while they are sitting in memory, but on the way to and from the memory.

Our attack will work regardless of the source of the error. Once we fetch a bad value into a local variable (typically implemented as a register in the processor), it doesn't matter whether the value became bad on to way from the processor to the cache, on the way from the cache to the memory, while sitting in main memory, on the way main memory to cache, or from cache to processor. All that we need is a local Java pointer variable containing slightly bad data.

Given the rarity of memory errors, an attack based on naturally occurring errors would have to attack many machines at once, hoping to catch a cosmic ray in one of them. This could be done by hiding the attack in an application program that is loaded on many machines. Be-

cause the attack requires very large amounts of memory to operate efficiently, the application in which it's hidden would itself have to be a memory hog. Fortunately for the attacker, few users are surprised these days when applications use hundreds of megabytes to accomplish trivial tasks.

Attacks on Static RAM

New generations of SRAMs are increasingly susceptible to memory errors [17]. SRAM error rates are orders of magnitude higher than DRAM error rates [6]. SRAMs are used for cache memory, often on the processor chip itself. Error detection is essential.

Our exploit should work against the data cache, although we have not measured it. In this case, we still need to allocate tens or hundreds of megabytes rather than just the cache size. The program address space should be large so that a flip in the maximum number of bits can be used.

7 Physical fault injection

If the attacker has physical access to the outside of the machine, as in the case of a smart card or other tamper-resistant computer, the attacker can induce memory errors. We considered attacks on boxes in form factors ranging from a credit card to a palmtop to a desktop PC.

We considered several ways in which the attacker could induce errors.[4]

Alpha particles are helium nuclei that are typically formed as a byproduct of radioactive decay of heavy elements. Obtaining an alpha-particle source from a scientific supply house might not be too difficult, or one could obtain a weak source by taking apart a smoke detector. However, alpha particles don't penetrate even a millimeter of plastic very well; historically, when alpha particles have been a significant source of memory errors it has been when radioactive sources have contaminated the chip packaging material itself. Alpha particles might be used to attack a computer in the form factor of a credit card, but anything thicker should be resistant.

Beta rays are high-energy electrons. They interact sufficiently strongly with plastic and metal packaging material that beta rays resulting from decay of radioactive nuclei would not be useful to an attacker.

X-rays or other high-energy photons might penetrate the packaging material, but interact weakly with DRAM circuitry – they simply don't have enough energy per particle. A dentist's X-ray or an airport baggage scanner would be very unlikely to induce memory errors. A "hard" (very high energy) X-ray source might possibly do the job.

High-energy protons and neutrons, such as those produced by large particle accelerators, are similar to those cosmic rays that penetrate the atmosphere, and interact similarly with DRAM chips. Such accelerators are often used to test the resistance of electronic components to cosmic radiation, especially components to be used on aircraft and spacecraft. Few attackers — indeed, few nation-states — have access to such accelerators. However, an Americium-Beryllium source (such as is used in oil exploration) produces neutrons that could very likely induce memory errors [13]. Access to such sources is regulated; an attacker could gain access by purchasing a small oil-drilling company, or by becoming employed at such a company.

Infrared radiation produces heat, and it is well known that electronic components become unreliable at high temperatures.

Since we lacked the time or inclination to learn the oil-drilling trade, we decided to use heat. We induced memory errors in a desktop PC by opening the box and shining light on the memory chips. We used a clip-on lamp with a flexible gooseneck, equipped with a 50-watt spotlight bulb.

At first we varied the heat input by varying the distance of the bulb from the chips. At about 100 degrees Celsius, the memory chips start generating faults. We were able to control the temperature so that errors were introduced in at most ten words, with errors in about 10 bits per word.

As we were fine-tuning this experiment, we found that introducing large numbers of memory errors would often cause the operating system not only to crash, but to corrupt the disk-resident software so that reboot was impossible without reinstallation of the operating system. To solve this problem, we arranged to boot Linux from a CD-ROM, without relying on the magnetic disk at all. The attacker would not have this luxury, of course; he would have to flip just a few bits the very first time.

For a successful exploit we wanted finer control over the temperature, so we controlled the lamp wattage with a variable AC power supply, and put the spotlight about

[4]We gratefully acknowledge a useful discussion with Dr. Eugene Normand [12] that helped rule out several classes of attacks.

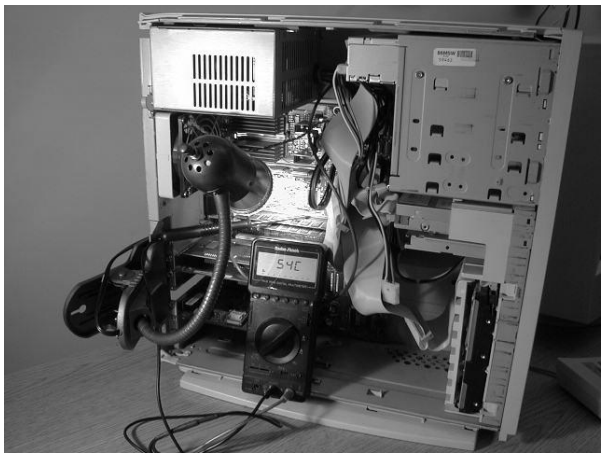


Figure 3. Experimental setup to induce memory errors, showing a PC built from surplus components, clip-on gooseneck lamp, 50-watt spotlight bulb, and digital thermometer. Not shown is the variable AC power supply for the lamp.

2 centimeters from the memory chips. We found a gradual rise in temperature in the region of 80–100° Celsius would cause isolated, random, intermittent soft failures in the memory. As section 5 explains, we expected that if we can induce isolated errors, the probability of a successful attack on the IBM JVM before the machine crashes is 73.3%.

This attack was successful. We ran one trial against each of the IBM and Sun JVMs, and each trial allowed us to circumvent the type system and take over the JVM. It takes about one minute to heat the memory in a successful exploit.

A real attacker would not have the luxury of opening the box and focusing just on the memory; it would be necessary to apply heat from the outside. For a palmtop or notebook-computer form factor, it might be possible to apply a focused light at just the place on the outside of the box under which the memory chips sit. For a desktop PC, this would be impossible; the attacker would have to heat the entire box (in an oven, or by blocking the cooling fan), and we don't know whether the memory would become unreliable before other components failed. A high-wattage AMD or Intel P4 processor would likely fail before the memory, but a low-wattage VIA C3 would not heat up as quickly as the memory [16].

It might also be possible for the attacker to heat specific memory chips by exercising them; the CMOS latch and datapath sections of the memory consume power mostly when changing state.

8 Countermeasures

Parity checking to detect single-bit memory errors, and more sophisticated error-correcting codes (ECC) to correct single-bit errors and detect multiple-bit errors, have been known and deployed for decades. The cost is small: to implement detection of 1-bit and 2-bit errors, it is sufficient to use 72 bits to represent every 64-bit word of memory, a memory overhead of 12.5%.

However, many or most mainstream desktop personal computers are sold without memory error detection hardware. One possible explanation is the price competition and low profit margins in the commodity PC business. If memory chips account for a quarter of the cost of a PC, and error detection adds a 12.5% overhead to the cost of the memory, then error detection adds a 3% overhead to the cost of the entire box; this is likely to be larger than the profit margin of the PC assembler/reseller.

Static RAM (SRAM) used in cache memory can also be a source of memory errors. Fortunately, in a typical desktop PC the cache may be on the processor chip, where there is no means or incentive for the assembler/reseller to omit ECC. Unfortunately, not all processors include ECC in cache datapath.

A fairly effective and obvious defense against our attack is to use a PC with ECC memory. However, a typical ECC design is meant to protect against naturally occurring faults, not against a coordinated attack. Therefore, there are additional considerations.

Multiple-bit errors. ECC memory can detect all 1-bit and 2-bit errors. The probability that a bit flips in the memory should be extremely small. Otherwise, we may have bit flips in the control space of the applet, and hence the applet may crash. For the adversary to successfully take over the virtual machine, the adversary should create a multiple bit error without creating 1-bit and 2-bit errors. Even, the probability of a 1-bit error is small. Thus, if single-bit errors are rare and uniformly randomly distributed, then the likelihood of a 3-bit error without ECC detecting any 2-bit or 1-bit errors is vanishingly small. However, ECC itself cannot provide a complete defense.

Total datapath coverage. Our attack works regardless of where on the datapath the error occurs. If there is a bus between processor and memory that is not covered

by error detection, then the attacker can attempt to induce errors in that bus. It is not sufficient to apply ECC just within the memory subsystem. Only a few high-end x86-compatible processors handle ECC on the processor chip [17].

Logging. Experts have long recommended logging of errors — even the single-bit errors that are automatically corrected by ECC hardware — so that patterns of problems can be detected after the fact. However, many operating systems do not log errors; this has made it difficult to diagnose problems [11].

To defend against attacks by heat or other means of inducing errors, the logging system must be able to react to a substantial increase in the number of errors. If several errors are detected in a short period, it would be wise to assume that the system is under attack, and to shut down — or at least to disable untrusted software that might contain implementations of our attack.

However, if a 3-bit or 4-bit error can be induced before very many 1-bit and 2-bit errors occur, then logging will not be successful: the attack will succeed before logging detects it. For a strong defense, more than 2-bit errors need to be detected, which can be done by increasing the number of ECC (overhead) bits in the memory.

9 Conclusion

Allowing the attacker to choose the program to be run alters many of the assumptions under which error-protection mechanisms are designed. Virtual machines that use static checking for protection can be vulnerable to attacks that exploit soft memory errors. The best defense is the use of hardware error-detection and correction (ECC), with software logging of errors and appropriate response to unusual patterns of errors.

Acknowledgments

We would like to thank Yefim Shuf, David Fisch, Michael Schuette, Eugene Normand, Peter Creath, Perry Cook, Brent Waters, Lujo Bauer, Gang Tan, Tom van Vleck, Crispin Cowan, Ed Felten, Jim Roberts, and Karthik Prasanna for their help in various stages of the project.

References

[1] R. Anderson and M. Kuhn. Tamper Resistance - a Cautionary Note. In *Proceedings of the Second Usenix Workshop on Electronic Commerce*, pages 1–11, Nov. 1996.

[2] R. Anderson and M. Kuhn. Low cost attacks on tamper resistant devices. In *IWSP: International Workshop on Security Protocols, LNCS*, 1997.

[3] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults. *Lecture Notes in Computer Science*, 1233:37–51, 1997.

[4] S. Borman. Understanding the IBM Java garbage collector. www-106.ibm.com/developerworks/ibm/library/i-garbage2/, Aug. 2002. web page fetched October 8, 2002.

[5] C. Colby, P. Lee, G. C. Necula, F. Blau, K. Cline, and M. Plesko. A certifying compiler for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*, New York, June 2000. ACM Press.

[6] A. Corporation. Neutrons from above: Soft error rates Q&As. Technical Report http://www.actel.com/appnotes/SER_QAs.pdf, Actel Corporation, July 2002.

[7] D. Dean, E. W. Felten, and D. S. Wallach. Java security: From HotJava to Netscape and beyond. In *Proceedings of 1996 IEEE Symposium on Security and Privacy*, May 1996.

[8] S. Drossopoulou and S. Eisenbach. Describing the semantics of Java and proving type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.

[9] S. Drossopoulou, T. Valkevych, and S. Eisenbach. Java type soundness revisited. Technical report, Imperial College London, Sept. 2000.

[10] G. McGraw and E. W. Felten. *Securing Java*. John Wiley & Sons, 1999.

[11] E. Normand. Single event upset at ground level. *IEEE Transactions on Nuclear Science*, 43:2742, 1996.

[12] E. Normand. Boeing Radiation Effects Laboratory, personal communication, Oct. 2002.

[13] E. Normand. Boeing Radiation Effects Laboratory, e-mail, Oct. 2002.

[14] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, C. J. Montrose, H. W. Curtis, and J. L. Walsh. Field testing for cosmic ray soft errors in semiconductor memories. *IBM Journal of Research and Development*, 40:41–50, Jan. 1996.

[15] D. Patterson. personal communication, Oct. 2002.

[16] M. Schuette. Enhanced Memory Systems Inc., e-mail, Nov. 2002.

[17] M. Schuette. Enhanced Memory Systems Inc., personal communication, Sept. 2002.

[18] T. Tso. random.c – a strong random number generator, 1994. drivers/char/random.c in Linux 2.4.19 source tree.

[19] D. von Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 119–156. Springer, 1999.

[20] J. Xu, S. Chen, Z. Kalbarczyk, and R. K. Iyer. An experimental study of security vulnerabilities caused by errors. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN-01)*, July 2001.

[21] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, B. Chin, M. Nicewicz, C. A. Russell, W. Y. Wang, L. B. Freeman, P. Hosier, L. E. LaFave, J. L. Walsh, J. M. Orro, G. J. Unger, J. M. Ross, T. J. O'Gorman, B. Messina, T. D. Sullivan, A. J. Sykes, H. Yourke, T. A. Enger, V. Tolat, T. S. Scott, A. H. Taber, R. J. Sussman, W. A. Klein, and C. W. Wahaus. IBM experiments in soft fails in computer electronics (1978-1994). *IBM Journal of Research and Development*, 40:3–18, Jan. 1996.

A Defeating address obfuscation in IBM's JVM

In our attack, it helps to have the address of an object so that we can design an an optimal layout to trap bit flips. If an applet can learn the address of an object, that in itself is not bad, but it may make other attacks easier. Dean et al. used the *hashCode* function to determine the object address in their attacks [7]. To defeat such attacks, modern JVMs try not to expose the address of an object. IBM JVM uses a cloaked object address in its *hashCode*. In this appendix, we show that IBM's cloaking method is ineffective.

HashCode function. Given an object, the Java specification requires that the *hashCode* method on the object return the same integer on every call. This method is provided to provide a good hash function for implementing hash tables. To reduce the number of collisions, it is desirable that given two objects, their hashCodes are different. One implementation is to return a pseudo-random number for each object and store the number in the header of the object. Another implementation, one that saves space in the object header, is to convert the internal object address into an integer. This is a typical implementation, and works if the object address is the same over its lifetime. In this implementation, if the object were to move during its lifetime due to compaction or due to a copying garbage collector, it is required to store the hashCode in the object header.

A.1 Hashcode Implementation in IBM's JVM

We used GDB to reverse engineer IBM's hashCode function implementation. The hashCode function in the IBM JVM is implemented as:

```
A  =  2 * sp + clock()
B  =  2 * sp + time() - 70
hashCode (address) {
   t1 = address >>  3
   t2 = t1 ^ A
   t3 = (t2 << 15) | (t2 >> 17)
   t4 = t3 ^ B
   t5 = t4 >> 1
   return t5
}
```

The JVM computes two global constants A and B during its initialisation. These constants are used in the computation of hashCode. In the above code, *sp* refers to the stack pointer at the entry of the function where A and B are computed. *Time* is the number of seconds elapsed since January 1, 1970; *clock* is the processor time used by the JVM since the start of the current process.

Apparently, the purpose of all the shifting and XORing is to obfuscate the address of the object. Exposing the address of an object is a bad security practice. Any bug in the type system or in the byte-code verifier coupled with the ready availability of the address might make the system vulnerable to attack [7].

Even without knowing the clock and time values, we can find if two objects' addresses differ by examining their hashCodes. If c is a constant, a_1 and a_2 differ in a bit *if and only if* $c \oplus a_1$ and $c \oplus a_2$ differ in a bit. Similarly, if both a_1 and a_2 are shifted circularly (as in the computation of t_3) or shifted right to remove constant bits (as in the computation of t_1—all objects in the JVM are allocated on an 8-byte boundary), then the resulting values a'_1 and a'_2 will differ in just one bit at a known position. This implies that for all bits, except bit 20, the object addresses differ in a bit *if and only if* the corresponding hashCodes differ in the corresponding bit. Bit 20 is the bit that is lost in the computation of t_5.

A.2 Obtaining the layout

To maximize our success probability, we base our attack on the object with the maximum number of cousins. We now describe an algorithm to obtain a layout which optimizes the cousin number of the object of type A. IBM's JVM implements a mark-and-sweep garbage collector [4]. We perform the following steps :

- Allocate a large number of objects b_1, b_2, ... each of type *B*.
- Compute the cousin number of each object, by flipping each bit in its hashCode, one at a time, and seeing if we have an object with the resulting hashCode. Choose the object β with the maximum cousin number; let the address of β be *x*.
- Deallocate the object β. This is done by setting all variables pointing to it to *null*. (Java does not have a *free* function.)
- Call the garbage collector; the address *x* is added to front of the free list because of mark-and-sweep garbage collector. The garbage collector is called by invoking the method *System.GC*. This JVM puts objects on the free list only if they are larger than 512 bytes (it relies on compaction to reclaim smaller objects), but our objects are large enough.
- Allocate an object α of type *A*. The memory manager reuses the address *x*, especially because objects of types *A* and *B* are of the same size.
- Set each field of type *A* of each object to α. Set the field *b* in object α to one of the *B* objects.

We thus have a layout where the cousin number of the object α (type *A*) is optimal. This layout maximizes our success probability.

A.3 Completely cracking the hashCode

For our attack, we did not need to find the exact *clock*, *time*, and *sp* values used in the hashCode function. But it is possible to (almost completely) undo the obfuscation. A Java applet may call System.currentTimeMillis() itself; it won't get the same value as the JVM did when initializing the constants *A* and *B*, but it will get a value that is delayed in a predictable way from the original call, especially if the JVM was invoked specifically to immediately load and execute only this applet. The applet can't call *clock()*, but this system call (on our Linux) is measured in increments of hundredths of a second, and the number of CPU centiseconds to initialize the JVM has only about 11 possible values (9 to 19).

The stack-pointer value is predictable too: in Linux, it depends only on the total sizes of the stack frames from the entry of *main* to the initialization of *A* and *B*—this is a constant that can be easily learned from reverse engineering of the JVM—and on the size of the Unix *argv* and environment string. The size of the environment string is unknown, but can be expected to be less then 10 kbytes.

Based on the limited randomness available in the sources of *A* and *B*, and given the applet's ability to allocate several objects (which will often be placed at consecutive addresses) and then query their hashCodes, we can find the values *A* and *B* with high confidence, or at least restrict them to a small set of possible values.

Even bit 20 of the address, which seems irretrievably lost in the computation of t_5, can be recovered by other means: objects allocated early will likely be in the first megabyte of process address space (bit 20 equals zero), and objects allocated later will be in the second megabyte (bit 20 equals one), and so on. The transitions can be observed in bit 19, and then the transitions in bit 20 can be inferred.

A.4 Conclusion

IBM's implementation of hashCode fails to provide effective obfuscation of the object address. The implementors would have done better to use a more effective source of randomness (such as /dev/random on Linux [18]), and to use a transformation that is less easily reversible than XORing with a constant. Alternatively, the JVM could have returned a random number and stored the number in the header.