Olympiads in Informatics, 2021 Vol. 15, 37–52

DOI: 10.15388/ioi.2021.04

Security of Grading Systems

Martin MAREŠ
Department of Applied Mathematics, Faculty of Mathematics and Physics, Charles University
Malostranské nám. 25, 118 00 Praha 1, Czech Republic
e-mail: mares@kam.mff.cuni.cz

Abstract. Programming contests often employ automatic grading of solutions. Graders need to run potentially malicious code, which brings many security issues. We discuss various attacks on grading system security and suggest counter-measures.

Keywords: automatic grading, security, sandbox, covert channels.

1 Introduction

Many of the world's programming contests are based on *automatic grading* (evaluation) of programs submitted by the contestants. Basically, the grading process takes a submitted *solution,* compiles it, executes it on a set of *test inputs,* and checks if the produced outputs are correct. Execution of the program is usually subject to a *time and memory limit,* so that full score is awarded only to efficient solutions. A typical example of such contest is the International Olympiad in Informatics (IOI), but many others exist.

The *grader* usually forms a part of a larger *contest system,* which also handles distribution of task statements, collection of submissions, displaying of the rank list etc.

Not all contestants play fair. Some of them attempt to cheat in order to gain an unfair advantage. Cheating can involve seeking forbidden advice from outside sources, but also breaking integrity of the contest system. We will study the latter type of cheating.

We will focus on the grader, because this is the component which runs potentially malicious code provided by the contestant. We will study various security issues pertaining to graders. We will discuss various possible attacks on security of grading. We will propose measures to thwart the attacks, or at least to mitigate their effects.

We are not the first to study grading security. Early research on this topic is summarized in the survey by Forišek (2006) and in the paper by Sims (2012). However, the landscape of contest systems has greatly changed since that time.

Contest systems now routinely employ advanced sandboxes based on kernel namespaces and control groups – for example Isolate by Blackham and Mareš (2012). Peveler *et al.* (2019) propose using full-edged containers to encapsulate untrusted code. New security features are added to the underlying operating systems. New types of contest tasks and new programming languages are introduced. These ask us to extend the traditional view of what a solution is supposed to do – for example, runtimes of several languages need multiple threads of execution. All these news bring their security challenges.

Several security problems mentioned by Forišek are no longer relevant (we will discuss these in section 2). Some continue to be abused in even more creative ways (denial-of-service attacks described in section 3, covert channels via grader feedback in section 5.2). Most importantly, many new types of attacks have surfaced (sections 4 to 7).

Some of the attacks were never published, but circulated in oral communication in the rather small communities of contest organizers and contest system developers. We had the luck of working in the International Technical Committee of the IOI, where many of these communities intersect. This paper is our attempt at making the issues more widely known, so that they could be avoided in as many contests as possible.

Still, our ambitions fall short of covering all the contests in the world. We will focus on IOI-like contests and graders running on the Linux operating systems. Our findings probably apply to other contests to some extent.

2. Obsolete attacks

We will start our exposition with several types of "obsolete" attacks. These were quite popular in the past (and mentioned in Forišek's survey), but they are no longer possible in current contest systems.

2.1. *Timing*

Contest systems need to measure execution time in order to enforce time limits. This is usually done using CPU time accounting provided by the operating system. The Linux kernel keeps track of total run time of each process separately. In theory, whenever a process is scheduled to a CPU for some time slice, the duration of the slice is added to the run time of the process.

However, reading the system timer at every context switch was traditionally considered too slow. Because of that, statistical sampling was used instead: At every tick of the system clock, the tick was charged on the currently running process.

This is reasonably accurate if the time slices are long (relatively to the ticks), but much less accurate with rapid context switches. Most importantly, as the sampling is not random, it can yield misleading results if the behavior of the processes is correlated with the system clock.

Indeed, it is possible to construct a program which uses carefully planned pauses synchronized with the system clock to avoid being caught running. Whenever the timer tick interrupt arrives, the program is sleeping. This way, the measured run time will be close to zero, even though the program could have been running for the most of the time.

This technique was described by Tsafrir *et al.* (2007), but at that time it was already well known in the community of kernel developers.

In early 2007 (kernel 2.6.13), Linux switched to the Completely Fair Scheduler. Timer-based sampling was abandoned in favor of reading the hardware clock explicitly. The new scheduler makes the class of attacks described above ineffective. We refer the reader interested in details to a survey of Linux scheduling by Iskhov (2015).

Even with the new scheduler, run time measurements cannot be perfectly precise. Most importantly, parallel processes influence each other in various ways. Temporarily giving the CPU to another process often flushes the caches. Processes running concurrently on multiple cores compete for memory bandwidth and sometimes also for access to level-3 cache. These effects were thoroughly analyzed by Mareš (2011) and the conclusions are still valid.

2.2. *Time-of-check to Time-of-use Race Conditions*

Early contest systems sandboxed graded processes using the `ptrace` syscall.[1] This is a mechanism intended chiefly for use by debuggers. It allows to stop a program whenever it attempts to make a syscall. A supervising process (a debugger, or in our case a sandbox manager) can then inspect the arguments of the syscall and decide whether the syscall should be allowed. Depending on that, the process is either allowed to continue or aborted.

At the first sight, this approach seems to be straightforward: we can enumerate a short list of "safe" syscalls and forbid all the others. However, even very simple functions in standard system libraries use a surprising variety of syscalls internally. All these must be included in the safe list, so it is actually quite long and it is no longer easy to argue that all syscalls on the list are safe under all circumstances.

More importantly, this approach is effective on single processes, but once we have two processes or threads sharing memory, it becomes fundamentally insecure. Suppose that a thread A invokes the `open` system call to open a file. One of the arguments of `open` is a pointer to the file name, stored as a string in memory. The thread is stopped by `ptrace`, the sandbox manager checks that the file name is innocent (e.g., it is the input file of the task). So it decides to let the thread A continue. At that time, thread B wakes up and overwrites the memory by a different file name. So the actual execution of the `open` syscall accesses the other file.

[1] *Syscall* is a system call from the user space to the kernel. See Kerrisk *et al.* (2012) for a description of the syscall interface.

This is a typical case of a so-called TOCTOU (time-of-check to time-of-use) vulnerability. It is almost impossible to avoid in sandboxes based on filtering of system calls. So these sandboxes are suitable for single-threaded programs only.

Fortunately, current contest systems mostly switched to sandboxes based on kernel namespaces. Instead of limiting the set of available operations, they limit the reach of these operations. For example, it is permitted to create network sockets, but the socket can connect only to other sockets within the same sandbox. This approach was pioneered in contest systems by Mareš and Blackham (2012), see their paper for description of one such sandbox and further discussion.

Namespace-based sandboxes avoid the TOCTOU problem, because the kernel reads the parameters from the user memory only once.

Still, the discussion in this section stays relevant. Some container supervisors (e.g., `systemd-nspawn`) apply system call filtering as an additional security measure. This typically involves the `seccomp` mechanism in the kernel (see Edge (2015)). It uses either a simple bitmap of permitted syscalls numbers, or a program for the BPF virtual machine. Since the BPF program can access only direct arguments of the syscalls and not contents of user memory, the TOCTOU issue does not occur. The problems with syscalls required by the standard libraries still remain, though.

3. Denial-of-Service Attacks

The easiest type of attacks on most computer systems are so-called *denial-of-service (DoS) attacks*. The attacker tries to consume as much resources as possible, reducing availability of the system to other users. We will discuss various ways of mounting a DoS attack on a grading system.

During on-site contests, DoS attacks happen rarely and usually by mistake – there is nothing to be gained. They occur more frequently in Internet contents, and sometimes also in practice sessions of on-site contests where the contestants try to push the system to its limits.

3.1. *Execution Time*

Submitted programs can consume various resources during their execution. It is trivial to write an infinite loop, infinite recursion, or an infinite loop allocating memory. All these are easily stopped by time and memory limits enforced by the sandbox.

Less frequent cases involve so-called fork bombs – programs iterating the `fork` syscall to create new processes. This usually leads to exponential growth in the number of processes running. Since individual processes are small, having a per-process memory limit does not help. Instead, a global a memory limit on all processes together must be imposed (e.g., as done by Isolate via process control groups). Alterna-

tively, we can limit the number of processes running inside the sandbox (e.g., by the `setrlimit` syscall).

Last, but not least, the program can fill up disk space by generating large files. Limiting file size using `setrlimit` is not enough, because multiple files can be created. A disk quota should be configured instead for the user ID under which the sandbox runs.

3.2. Compilation Time

When a contestant submits the source code of their solution, it has to be compiled first. Even though contest systems developers concentrate mainly on security of running the solution, many attacks can be performed in the compilation stage, too.

First of all, it is easy to write a short program whose compilation takes an arbitrarily long time. For example by chaining C++ templates:

```
#include <map>
using namespace std;

typedef map<int,int> M1;
typedef map<M1,M1> M2;
typedef map<M2,M2> M3;
// ...
typedef map<M15,M15> M16;

int main() { M16 tmp; return tmp.size(); }
```

This construction was already mentioned by Forišek. We confirm that it still works with current compilers (e.g., GCC 8.3.0), even though the template chain must be made longer to produce significant delay. (It should be no surprise as the C++ templates are known to be Turing complete, see for example Veldhuizen (2003).)

The compiler can be also trapped by `#include "/dev/zero"` – a special file containing an infinite sequence of zero bytes. GCC desparately looks for an end-of-line character, which would terminate the infinitely long line, and buffers the contents of the line in memory. This usually exceeds the memory limit sooner than the time limit.

It is also possible to include `/dev/urandom` instead – a source of infinitely many cryptographically strong pseudo-random numbers. The pseudo-random generator is sufficiently slow to make GCC exceed the time limit first.

Finally, the compiler can be also made to generate a very long output file. A particularly easy method is to create a huge static array with only the last element initialized (a completely uninitialized array would not be stored explicitly):

```
#define N 1000000000
char huge[N] = { [N-1] = 1 };
```

4. Attacks on In-Process Graders

At most programming contests, solutions read their input and write their output as files. The International Olympiad in Informatics uses a different interface since 2010: the task specifies an API between the solution and the *grader*.[2] The contestants have to write a program which conforms to this API – they implement functions called by the grader, and they can call other functions provided by the grader. The same approach is also used for interactive tasks at other contests.

Let us consider the following task as an example: The first player thinks of a secret integer S between 0 and 10^6. The goal of the second player (implemented by us) is to guess this integer using at most 20 queries of the type "Is the integer S less than x?".

The API consists of two functions:

- `int play()` – implemented by the contestant. Plays the game and returns the guessed integer.
- `bool less_than(int x)` – implemented by the grader. Returns if the secret number is less than the given x.

During compilation, the solution is linked with the grader. The grader contains the entry point of the whole program (the function `main()`). It generates the secret integer and calls `play()` defined in the solution. The `play()` calls `less_than(x)` back in the grader, which compares the x with the secret number stored in the grader and answers accordingly. At the end, `play()` returns to the grader, which compares the result with the secret number and stops the program. The exit code of the program can be used to signal whether the solution was correct.

In more complex tasks, there can be an input file read by the grader, and the grader can write the output to an output file, so that it can be checked by the other parts of the contest system.

The API-based interface has numerous advantages. It hides implementation details from the contestants. This can be important in languages, where efficient access to files requires various kinds of trickery (Java, but to some extent also streams in C++). In case of interactive tasks, it relieves the contestants of thinking about flushing output buffers properly. Furthermore, the low overhead of function calls (as opposed to inter-process communication) makes it easy to implement interactive tasks with lots of interaction.

Unfortunately, the security of this approach is very bad. We have a system with two components: a trusted grader and a completely non-trusted solution. We need to establish some kind of a security boundary between the components, so that the solution cannot influence the grader in unintended ways. A shared address space within a single process makes for a very poor security boundary. The solution can access all the data of the grader and it can even modify its code.

[2] Technically speaking, the term "grader" is misleading here. It is not the complete grading component of the contest system, but only a small piece capable of checking correctness in the particular task. However, we will stick to the term grader as used in IOI jargon for this chapter.

Graders used in past IOIs took effort to make the attacks less likely to succeed (and unintended crashes caused by bugs in solutions less likely to happen). They used cryptic hard-to-guess names for their internal functions and variables. They were writing a special secret string (a "cookie") at the start of the output file, which makes it easy to detect that the output was not written properly by the grader. The cookie itself was stored in an obfuscated form inside the grader. Also, the input file was obfuscated and decoded by the grader.

All these measures certainly make attacks harder to succeed and easier to detect. Still, from the security point of view, they are typical examples of "security by obscurity". And indeed, these were not completely effective in practice and some successful cheating attempts were discovered.

4.1. *Exchanging Library Functions*

Let us consider the typical case of a C++ solution, linked statically (as done by the CMS contest system at the IOI; the situation would be similar, though different in details, with dynamic linking). The linker combines four pieces of code: three object files (the solution, the grader, and a piece of startup code provided by the compiler) and the standard C library. The library is a collection of object files, usually one file per library function or a small group of functions.

The linker collects the initial object files and looks at their dependencies. They have the form of symbol references – an object file can ask for an arbitrary symbol (a named function or external variable). The linker tries to find the definition of this symbol in the other object files first. Only if it is not defined there, the linker inspects the library, finds an object file defining that particular symbol, and adds this object file to the current set of object files. The new object file can reference further symbols from the library and so on. The whole process is repeated until all symbol references are resolved.

Even though the grader uses cryptic names for all internal symbols, it still needs to call functions from the standard library (e.g., to read from files) and these have fixed names. But library functions can be easily interposed by the solution, since symbol definitions in other object files have priority over those in the library. For a concrete example, when the grader calls `write()` and there is `write()` defined in both the solution and the library, the grader will call the version from the solution.

This makes it possible for the solution to modify the output of the grader, which already contains the secret cookie. Of course, the solution must write the output using a different library function (e.g., `writev()`), or to invoke a syscall directly.

This attack can be mitigated by using two-step linking: first link only the grader with the standard library, but tell the linker to produce another object file as a result. In the second step, link this object file with the solution and again with the standard library. All calls to library functions in the grader are already resolved in the first linking, so the second linking cannot re-bind them to symbols from the solution.

4.2. *Rolling Back Grader State*

There is a much more serious attack on graders which keep some internal state to check correctness. In our toy example with guessing of numbers, the grader keeps a counter of queries, so that it can reject solutions asking more than 20 queries. The attacker could be interested in manipulating this counter to ask more queries with impunity.

Gaining enough information about internals of the grader to locate the counter in memory is hard to do under time pressure of the contest. Especially if the value of the counter is obfuscated or the grader stores multiple copies of the counter.

However, there is a much easier way: Simply make a copy of all data belonging to the grader and copy it back later. This allows to make a snapshot of the complete state of the grader and revert the grader to that state later.

Still, finding out what memory belongs to the grader is non-trivial. But the solution can easily identify its own memory, so it can copy all memory except its own. A good starting point is `/proc/self/maps`, which lists all virtual memory areas in the address space of the current process.

It is easy, but it can be made even easier if the solution is allowed to create multiple processes. The standard way of creating a new process is the `fork()` syscall – it creates a clone of the current process, which differs only by the process ID and the return value of `fork()`. So the solution can fork a new process, which has a copy of the grader's state. Then it can run a part of its computation inside that process, pass the results back to the main process, and exit. The main process still keeps the original state of the grader, not aware of any queries which were made in the new process. Since process creation is cheap, this can be easily iterated.

4.3. *Proper Design of Graders*

We described many security problems with in-process graders. In our opinion, attempts at fixing them are futile, because a proper security boundary between two machine-code programs inside the same process is impossible to maintain. We firmly believe that security by obscurity must be rejected and replaced by a mechanism that is secure by design.

In case of batch tasks (read input, then produce output), the grader can implement the task's API by reading and writing of files (or the standard input and output). All checking of correctness should be done outside the sandbox after the solution stops.

Interactive tasks are more subtle. If the amount of interaction is not too large, the grader can convert the API functions to communication over a pipe (or a UNIX-domain socket) with a separate grader process running outside the sandbox.

For tasks which require very intensive interaction, the latency of communication over a pipe can be prohibitively large. In such cases, a block of shared memory guarded by simple spinlocks can be used instead. The grader should copy the data to its local memory first to avoid TOCTOU issues mentioned in Section 2.2. This type of commu-

nication is more complicated and it needs further investigation. We are still convinced that security is worth the effort.

5. Covert Channels

In many contests, the solutions are tested on test data stored inside the grading system, but supposed to be unknown to the contestants. Under some circumstances, it is possible for the contestants to gain some information about the secret test data. This is a classic example of what security researchers call a *covert channel* between two components separated by a security barrier. We will describe several kinds of covert channels.

5.1. *Secrets Lying on the Disk*

Surprisingly often, secret input files, or even the correct outputs, could be found somewhere in the file system, readable to the solution. This was usually a result of bad configuration caused by human mistake. Namespace-based sandboxes make it much less likely to happen, since only an explicitly selected subset of the directory hierarchy is available inside the sandbox.

Execution environment of solutions is usually very restricted, but the compilation environment not much so. Compilers tend to require access to lots of unexpected files. It is therefore tempting to grant compilers read-only access to huge subtrees of the directory hierarchy. We will show that all information available to the compiler is effectively available to the solution, too.

Of course, if the compiler has access to a reference solution, the contestant's solution can simply use the C/C++ preprocessor to `#include` the reference solution. This however does not help to gain access to test data, because test data seldom conform to syntax of a C++ fragment, which could be included.

In such cases, file contents can be obtained using inline assembly. The GCC compiler does not produce machine code directly. It generates symbolic instructions and feeds them to an assembler (in case of GCC, it's the GNU Assembler, gas). GCC extends the C language by adding an `asm` statement, which can pass arbitrary strings directly to the assembler. This can be used to execute not only any machine instructions, but also arbitrary compiler directives (pseudoinstructions) of the assembler. One such directive is `.incbin`, which includes complete contents of a specified file byte by byte in the object file being produced.

Let us see a simple example, which embeds contents of `/etc/passwd` in the compiled program and prints it out when the program is run.

```
asm("\
   .data\n\
   start_hack: .incbin \"/etc/passwd\"\n\
   end_hack:\n\
```

```
   .text\n\
");

extern char start_hack[], end_hack[];

#include <unistd.h>
int main() {
   write(1, start_hack, end_hack - start_hack);
}
```

The `.data` directive switches to the data section, so that the contents of the file become part of the program's data. The `.text` directive switches back to the code section. The start and end of the file are demarcated by the `start_hack` and `end_hack` labels, which are accessible as external variables of array type in C code.

5.2. *Grader Feedback as a Covert Channel*

Some contests provide feedback on submissions, which allows contestants to fix errors and re-submit the solutions. All such feedback can be used as a covert channel, although of very small bandwidth.

Suppose that every submission is tested on 10 different test cases. The feedback for each test case includes the verdict (correct / wrong answer / time limit exceeded / runtime error), time used (in milliseconds) and memory used (in megabytes). Time limit is 5 seconds, memory limit 256 megabytes.

We estimate how much information about a single test case can be encoded in the feedback. We will ignore the verdict. We can generate an arbitrary memory usage between 1 and 256 megabytes, so we can encode 8 bits in that. Encoding information in execution time is more subtle as the value is influenced by measurement errors. According to Mareš (2011), random noise does not typically exceed small tens of milliseconds. So we will let the program run for a multiple of 100 ms and round the reported time to the nearest such multiple. This gives us 49 values between 0.1 s and 4.9 s, which can be used to encode 5 bits.

In this simple example, we can gain 13 bits of information about every test case per submission. The number of submissions is usually limited, so we cannot exfiltrate full test data from the contest system. But it is certainly possible to extract sizes of test inputs and other basic characteristics.

Because of this, contest organizers should put the amount of feedback under close scrutiny and try to minimize the effective bandwidth of the covert channel, while keeping the feedback useful to honest contestants.

Counter-measures can include providing only summary statistics like maximum execution time and memory over all test cases, and the histogram of verdicts (e.g., 3 times “passed”, 5 times “wrong answer”, and twice “time limit exceeded”).

Furthermore, task designers should avoid tasks with very small entropy of inputs.

5.3. */proc File System*

Finally, we mention one unexpected source of potential information leaks: the `/proc` file system. It is a virtual file system whose files contain information about currently running processes (hence the name) and also global information on the state of the system (e.g., how much memory is used for what purpose). This is where commands like `ps` or `top` obtain their knowledge.

Although some information on processes (e.g., the set of descriptors of open files) is available only to the owner of the process, a small subset is available to all users. The public subset includes arguments given to the program when it is executed.

Visibility of the arguments is considered well-known among UNIX programmers, so people traditionally avoid passing passwords and other sensitive strings as arguments. However, we would like to remind the authors of contest systems and competition tasks about this issue. Process arguments should be never used for passing any information which should stay secret from the contestants. This includes things like test case number if the grader of a test case is started as a separate program. A better choice is to pass the information in environment variables, which are not considered public.

Sandboxes based on process namespaces mitigate this problem, because the `/proc` file system inside the sandbox reports only the processes living in the namespace of the sandbox. In addition to processes and threads of the solution, this includes only the sandbox manager itself. The Isolate sandbox makes an extra effort to hide even the sandbox manager, so that sandbox settings passed as arguments are not visible.

6. Cross-Language Attacks

New programming languages are growing at a surprising pace. Contest organizers would prefer to make available more languages than C++. A popular choice is Python with the intention of making the contest more accessible to beginners. A typical problem with Python is its speed, or lack thereof. IOI-level tasks typically require strict time limits, so that it is possible to distinguish between solutions of different time complexity. But a time limit which allows an optimal solution in C++ to run with a generous margin of 200 %, is grossly insufficient for an optimal solution in Python.

Organizers of competitions often try to sidestep this issue by allowing a higher time limit for Python programs. This necessarily involves a rather philosophical question of fairness, which we will avoid here. Instead, we will demonstrate that any such configuration can be easily misused to run a C++ solution with Python time limits.

The idea is to compile a C++ program, embed the resulting binary in a Python program, which will reconstruct the binary and run it.

Let us show a concrete example. We take a test program `test.cpp`, compile it to an executable file and strip off all debugging symbols to save space:

```
g++ -O2 -s test.cpp -o test
```

Then we use Python to compress the executable file and encode it in Base64 format. Base64 consists purely of printable ASCII characters, which can be embedded in our Python "solution".

```
import zlib, base64
bin = open('test', 'rb').read()    # Read the binary
compr = zlib.compress(bin, 9)      # Compress it
b64 = base64.b64encode(compr)      # Encode it in Base64
print(b64.decode('us-ascii'))      # Print the result
```

The solution itself will decode the executable file, write it to the current directory and execute it from there:

```
import base64, zlib, os
b64="""... output of the previous program ..."""
compr=base64.b64decode(b64)        # Decode Base64
bin=zlib.decompress(compr)         # Decompress

with open('hack', 'wb') as f:      # Write the binary to a file
   f.write(bin)

os.chmod('hack', 0o777)            # Grant executable permission
os.system('./hack')                # Execute
os.unlink('hack')                  # Cover up tracks :)
```

This form of the attack works in contests which pass input and output using files or standard input and output. In contests with an IOI-style API, it is necessary to call Python functions of the API from the C++ program. To accomplish this, the C++ program should be compiled as a Python extension and loaded from the Python program using `import`. Technically, Python extensions written in C++ are dynamically linked libraries with a well-defined interface to the Python interpreter. They can export functions callable from Python, call Python functions, and access state of the Python interpreter. We have a proof-of-concept implementation of such attack, but it is too large to include in this paper. It is still possible to write it within the few hours of a contest.

There are multiple potential mitigations. Generous limits on source code size can be lowered. The basic form of our attack requires writing to files, so the whole filesystem can be mounted read-only (and standard output connected to a writable file staying outside the sandbox). Or the writable directory can be mounted with the `noexec` option, so that code cannot be executed from it.

These mitigations are not ultimately effective. Writing to files can be replaced by creating a virtual file residing in memory using the `memfd_create` syscall, which is available in Python since version 3.8. It is still necessary to specify a file name when executing the program, but the name can refer to an open file descriptor using `/proc/self/fd/`*N*.

It is tempting to forbid tricks of this kind in contest rules, but we suspect that there are many variants on this theme and some of them could look innocent, or at least as corner cases of the rules. Similar methods can be used for most, if not all, other pairs of languages – see the `AnyExec2C` tool by Lukeš and Šraier (2020).

7. Other Attacks

In this section, we would like to mention two attacks, which do not fall into more general categories.

7.1. *Using Threads to Increase Cache Size*

Contests often allow starting of multiple processes or threads, because it is required by runtime environments of some languages (most notably Java and the .NET runtime). In such cases, time and memory limits are imposed on the group of processes as whole, typically using Linux control groups.

Initially, it seems that splitting computations to multiple threads does not yield any advantage. Even though the threads can run in parallel on a multiprocessor (or multi-core) machine, the time limit will be applied to the sum of the run times of the threads.

Surprisingly, there are cases where using multiple threads can improve run time. The reason is that current processors contain cache memory, which is much faster than the main memory, but much smaller. The effects of caching are quite complex, because processors often use several levels of caching. So we sketch just a simple estimate. The largest level of the hierarchy on current PCs (the L3 cache) holds several megabytes of data and it is shared among a subset of cores. Relatively to access to the L3 cache, access to main memory is about 5 times slower and access to other L3 caches about 3 times slower.

Suppose that the working set of our program (the data it accesses frequently) exceeds the size of a single L3 cache by a small factor. If the program runs on a single core, it will have only the L3 cache of that core available, so the data will not fit in the cache. If we split the program to multiple threads, they can run in parallel on multiple cores and have access to multiple L3 caches. Even though access to remote L3 cache is slower than to the local L3 cache, it is still substantially faster than access to main memory.

This attack is hard to carry out and quite simple to stop: bind the sandbox running the solution to a specific core. This way, even if the solution starts multiple threads, all of them will be scheduled in alternating timeslices on that single core. Obviously, the same computation can be performed by a single sequential program, so there is no possible gain from parallelism. (Except for possibly simplifying control ow of an interactive program, but that is hardly a cheat.)

7.2. *Storing Data in Socket Buffers*

If a task uses particularly tight memory limits, contestants can find numerous ways of storing data outside the address space of the their process. Depending on the method of measuring memory use, these ways will or will not be counted. If the sandbox uses traditional UNIX resource limits (see the `setrlimit` syscall), they will not be. If it uses memory control groups, they will.

A particularly nice example is the socket buffers – for each socket (an abstraction of a network connection), the kernel keeps a queue of data which was accepted for transmission, but not yet received by the other end of the connection. For TCP connections, the size of the queue is controlled by the tunable parameter `/proc/sys/net/ipv4/tcp_rmem` and the default maximum size is 6 MiB. By establishing 17 TCP connections to itself, a solution can keep 100 MiB of data in socket buffers.

7.3. *Security Issues in Processors*

Security issues are not limited to software. In January 2018, the world was caught by surprise by publication of several hardware vulnerabilities in processors. The most severe of these vulnerabilities are known under the names Meltdown (Lipp *et al.* (2018)) and Spectre (Kocher *et al.* (2019)). Meltdown affects only Intel CPUs, some forms of Spectre apply to all processors which employ branch prediction. Other similar problems were discovered later.

The underlying principle of all attacks of this class is the presence of subtle, but measurable side-effects of instructions which were executed only speculatively, but later rejected, because the speculative assumption turned out to be false. This can be used to create covert channels across various kinds of security barriers. For example, the original proof-of-concept code for Meltdown allowed an ordinary process to access kernel data. Spectre can be misused by JavaScript programs running inside the web browser to steal cookies belonging to other sites.

After a huge effort, these vulnerabilities were mostly mitigated in software, most importantly in the Linux kernel. The mitigations successfully prevent covert channels between different processes at the cost of slowing down all programs slightly. Only in cases where the security boundary goes within a single process, like in web browsers, additional per-process mitigations are necessary.

In our opinion, contest systems developers need not be worried about this class of attacks. Work-arounds implemented in the Linux kernel are sufficient in all cases where untrusted code is sandboxed properly.

8. Conclusion

We surveyed many possible attacks on security of contest systems. Some of them rather theoretical, some of them already observed in real contests. We discussed defenses against the attacks. We will summarize our recommendations here.

The contest system should be properly separated to trusted parts (e.g., the grader) and parts running untrusted code (execution of contestant's solution, but also its compilation). All untrusted code should be carefully sandboxed, there should be a narrow and well-defined interface between the sandbox and the trusted code. Every sandbox should place limits on total time and memory consumption, but also on total disk space used.

In particular, it is crucial to avoid any mixing of grader code responsible for checking correctness with any contestant's code in the same process. We outlined possible solutions in section 4.3. The case of interactive tasks with intensive interaction needs further investigation.

All feedback given to constants provides a potential covert channel for exfiltrating information on secret test data. The exact form of feedback should be chosen carefully to minimize the bandwidth of this channel.

Discriminating between solutions based on their programming language turns out to be inherently problematic. Section 6 suggests some mitigations, but they are necessarily incomplete.

Computer security is a rapidly developing discipline full of surprises (cf. the security issues of CPUs in section 7.3). As the saying goes, attacks never become weaker with time. We must continue our study, investigate new ways of attacking systems, and build our systems to be even more resilient.

References

Blackham, B., Mareš, M. (2012). A new contest sandbox. *Olympiads in Informatics*, 6, 100–109.

Edge, J. (2015). A seccomp overview. In: *Linux Weekly News*, 2015-09-02. Available on-line at: `https://lwn.net/Articles/656307/`

Forišek, M. (2006). Security of programming contest systems. In: V. Dagienė, and R. Mittermeir (eds.), *Information Technologies at School*, 553–563. Available online at `https://people.ksp.sk/~misof/publications/copy/2006attacks.pdf`

Ishkov, N. (2015). *A Complete Guide to Linux Process Scheduling*. M.Sc. thesis, School of Information Sciences, University of Tampere, Finland. Available online at `https://trepo.tuni.fi/bitstream/handle/10024/96864/GRADU-1428493916.pdf`

Kerrisk, M. *et al.* (2012). *The Linux Man-Pages Project*. Available on-line at `http://www.kernel.org/doc/man-pages/`

Kocher, P. *et al.* (2019). Spectre attacks: Exploiting speculative execution. *2019 IEEE Symposium on Security and Privacy (SP)*. Available on-line at `https://ieeexplore.ieee.org/iel7/8826229/8835208/08835233.pdf`

Lipp, M. *et al.* (2018). Meltdown: Reading kernel memory from user space. *27th USENIX Security Symposium (USENIX Security 18)*. Available on-line at `https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-lipp.pdf`

Lukeš, S., Šraier, V. (2020). `AnyExec2C`. Available on-line at `https://github.com/exyi/anyexec2C`

Mareš , M. (2011). Fairness of time constraints. *Olympiads in Informatics*, 5, 92–102.
Peveler, M., Maicus, E., Cutler, B. (2019). Comparing jailed sandboxes vs containers within an autograding system. In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*.
Sims, R.W. (2012). *Secure Execution of Student Code*. Technical report of Department of Computer Science, University of Maryland. `http://honors.cs.umd.edu/reports/simspaper.pdf`
Tsafrir, D., Etsion, Y., Feitelson, D.G. (2007). Secretly monopolizing the CPU without superuser privileges. In: *USENIX Security Symposium 2007*. pp. 239–256. Available on-line at `https://www.usenix.org/legacy/event/sec07/tech/full_papers/tsafrir/tsafrir_html/`
Veldhuizen, T.L. (2003). *C++ Templates are Turing Complete*. DOI: 10.1.1.14.3670. Available on-line at `https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.3670`

M. Mareš is a lector at the Department of Applied Mathematics of Faculty of Mathematics and Physics of the Charles University in Prague, organizer of several Czech programming contests, member of the IOI Technical Committee, and a Linux hacker.