The Symbols of Operation

Chris Anley, Chief Scientist, NCC Group plc

Introduction

Countess Ada Lovelace wrote in 1843 about “the science of operations” - what we would today call Computer Science - and identified a problem that has been important in computer security since the advent of the modern computer, the potential for confusion between code (“the symbols of operation”) and data. The Countess mentions this in passing and mostly in reference to mathematical notation, but it’s striking that in her narrative, she moves directly into a description of the manner of separation of code and data within the mechanism of the Analytical Engine.

This difficulty of distinction between code and data - and the dangerous consequences when they become confused - is at the heart of exploits of “arbitrary code execution” vulnerabilities such as buffer overflows, and is very much a current topic of research within the Computer Science and Security communities.

In very recent years, with the rise of Large Language Models and in particular, Agentic AI systems, we are seeing another wave of security vulnerabilities whose root cause is this same confusion - between “the symbols of operation”, and the data they operate upon.

What the Countess Wrote

(Quote from the translation of the original French, into English, by Ada Augusta, Countess of Lovelace, of “Sketch of the Analytical Engine Invented by Charles Babbage By L. F. Menabrea of Turin, Officer of the Military Engineers”, taken from section “Notes by the translator, Note A”)

In abstract mathematics, of course operations alter those particular relations which are involved in the considerations of number and space, and the results of operations are those peculiar results which correspond to the nature of the subjects of operation.

But the science of operations, as derived from mathematics more especially, is a science of itself, and has its own abstract truth and value; just as logic has its own peculiar truth and value, independently of the subjects to which we may apply its reasonings and processes.

Those who are accustomed to some of the more modern views of the above subject, will know that a few fundamental relations being true, certain other combinations of relations must of necessity follow; combinations unlimited in variety and extent if the deductions from the primary relations be carried on far enough.

They will also be aware that one main reason why the separate nature of the science of operations has been little felt, and in general little dwelt on, is the shifting meaning of many of the symbols used in mathematical notation.

First, the symbols of operation are frequently also the symbols of the results of operations.

We may say that these symbols are apt to have both a retrospective and a prospective signification. They may signify either relations that are the consequences of a series of processes already performed, or relations that are yet to be effected through certain processes.

Secondly, figures, the symbols of numerical magnitude, are frequently also the symbols of operations, as when they are the indices of powers.

Wherever terms have a shifting meaning, independent sets of considerations are liable to become complicated together, and reasonings and results are frequently falsified.

Now in the Analytical Engine, the operations which come under the first of the above heads are ordered and combined by means of a notation and of a train of mechanism which belong exclusively to themselves; and with respect to the second head, whenever numbers meaning operations and not quantities (such as the indices of powers) are inscribed on any column or set of columns, those columns immediately act in a wholly separate and independent manner, becoming connected with the operating mechanism exclusively, and re-acting upon this.

They never come into combination with numbers upon any other columns meaning quantities; though, of course, if there are numbers meaning operations upon n columns, these may combine amongst each other, and will often be required to do so, just as numbers meaning quantities combine with each other in any variety.

It might have been arranged that all numbers meaning operations should have appeared on some separate portion of the engine from that which presents numerical quantities; but the present mode is in some cases more simple, and offers in reality quite as much distinctness when understood.

The operating mechanism can even be thrown into action independently of any object to operate upon (although of course no result could then be developed). Again, it might act upon other things besides number, were objects found whose mutual fundamental relations could be expressed by those of the abstract science of operations, and which should be also susceptible of adaptations to the action of the operating notation and mechanism of the engine.

Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent.

The Analytical Engine is an embodying of the science of operations, constructed with peculiar reference to abstract number as the subject of those operations.

The Problem, Part 1 - Buffer Overflows

The majority of modern computers follow the "von Neumann" architecture, where code - instructions to be executed - is stored in main memory, alongside the data the instructions operate upon. The ability to treat instructions as data makes assemblers, compilers, linkers and other programming tools possible - it is possible to write "programs that write programs".

Self-Modifying Code

Programs that write programs, and the subset, programs that modify themselves, have a long history in computing. The EDSAC, built in 1947, had no index register, so in order to operate on the elements of an array, a program would have to modify addresses within its own instructions. Similar mechanisms were used in subsequent designs such as the PDP-1, and in more recent history programming languages such as LISP, JavaScript and Python contain facilities to execute dynamic code fragments at runtime (via "eval()" and similar). Just-In-Time (JIT) compilers create a program's executable code at runtime, typically as a performance optimisation or to enable greater portability. JIT compilation mechanisms are used in almost all modern web browsers to improve performance, so if you're viewing this article in a web browser, you're almost certainly running self-modifying code right now.

If you've played a modern video game, you've probably been running an anti-cheat program that modifies the code of running processes in order to protect their integrity. Similarly, several modern Digital Rights Management systems make use of anti-debug and anti-tamper software, which in turn often rely on self-modifying code.

Buffer Overflow

Dynamic manipulation of code brings us to the buffer overflow; perhaps the most impactful security issue of the past 40 years. The first recorded buffer overflow exploit was part of the Morris Worm, released in 1988, exploiting a stack buffer overflow vulnerability in the Unix service 'finger' (resulting from the C standard library function 'gets()'). The issue had been publicly documented as early as 1972 when the Multics operating system security evaluations (commissioned by the U.S. Air Force) discussed the dangers of "unchecked array bounds" in system code, noting that "This can be used to inject code into the monitor that will permit the user to seize control of the machine".

The Common Vulnerabilities and Exposures (CVE) list project attempts to record all publicly-known information security vulnerabilities, across all hardware and software platforms. In the calendar year 2024, there were 40,303 vulnerabilities recorded in the list, which works out at approximately one vulnerability recorded every fifteen minutes. Around 10% of those (4,119) were buffer overflow or related memory corruption vulnerabilites, with the overwhelming majority of these leading to arbitrary code execution. In the past 10 years there have been roughly 40,000 of these issues, with each of those 40,000 issues typically representing a critical code execution vulnerability in a widely-used piece of software. By exploiting a buffer overflow, an attacker generally gains full control of all data accessible to a service, and frequently full control of an entire host.

Buffer overflows occur when too much data is copied into a memory buffer. The excess data overwrites whatever follows the buffer in memory. In the case of the 'stack', the data that is overwritten contains return addresses, function pointers, frame pointers and other data that defines the behaviour of the running program. By carefully selecting the values they write, an attacker can modify these pointers, resulting in the execution of code of their choice. In the case of the Morris worm, for example, the exploit consisted of an overly-long request to the 'finger' daemon that overwrote the 'saved return address' on the stack, which is a kind of bookmark that records the location the program should continue executing from, once the current function has exited. This caused the program to 'return' *into* the request itself, directly executing code within the request the worm had just sent.

Other forms of buffer overflow vulnerability exist, varying depending upon the location in memory and the specifics of the data modified when the buffer overflows; for example, heap overflows occur when a dynamically-allocated buffer on the heap is overflowed, resulting in heap management data such as linked-list pointers and indexes being overwritten with values the attacker has chosen, which in turn leads to memory 'write' operations occuring under the attacker's control.

An early mitigation to buffer overflows involved setting memory page protections on the stack to ensure that the stack became 'non-executable'. This directly prevents exploits like the Morris worm finger exploit, where the attacker causes the program to 'jump' directly into data they have supplied, however this mitigation can be bypassed in various ways:

- Not all processors support memory page protections, for example "Internet of Things" devices created using chipsets that lack these features can be built more cheaply that those with greater levels of protection.
- The attacker need not execute code on the stack. If the program stores attacker-supplied data in any (non-stack) area of memory that is executable, the attacker can cause their exploit to 'jump' to that region instead.
- The attacker need not execute code at all; they can simply modify data, changing the behaviour of the program.
- The attacker can cause the program to execute code that is already present in memory, supplying their own parameters and context to change the program's behaviour.

An early example of a technique to bypass to this 'Non Execute' (NX) or 'Data Execution Prevention' behaviour was 'ret-to-libc'. The 'libc' library is a library of standard functions provided by the C programming language, and includes many useful functions to create processes, manipulate files and so on. By causing the program to 'return to libc', the attacker can call a single function in this library, with parameters they control. So for example, the attacker could run a shell command using the libc function 'system()'. This is a crucial point in our discussion of the distinction between code and data - is a shell command code?

A subsequent refinement to this technique is ‘ROP’ (Return-Oriented Programming), in which the attacker overwrites multiple stack frames to achieve a series of controlled jumps into pre-existing code of their choice. Notably, the attacker need not call an entire function, as in the ret-to-libc technique, since the attacker fully controls the location the program will return to, they can return to any small section of code that achieves their purpose, even potentially jumping to a location within an instruction in some architectures. This allows the attacker to piece together a ‘ROP chain’ of multiple snippets of code (‘gadgets’) into an arbitrary sequence, ultimately resulting in the execution of entirely arbitrary code, once the codebase is large enough. Academic research in this area has confirmed that ROP attacks are Turing-complete, in other words, existing snippets of code provide a ROP exploit with the ability to perform any operation desired by the attacker.

This is extremely interesting in the context of the distinction between code and data. The naïve stack overflow exploit supplies data which overwrites control structures (also data), causing the path of execution of the program to ‘jump’ into a portion of the data supplied. This is conceptually easy to understand, and the obvious fix is - prevent the execution of ‘data’. The problem is that the flow of execution of the program is entirely controlled by data (saved return addresses and other function pointers), to the extent that by altering this ‘control flow’ data, we change what the program does. Due to the ease with which the small snippets of code (‘gadgets’) can be combined to form a Turing-complete language, we effectively create new, entirely arbitrary code. As the Countess might put it, the “figures” have become “the symbols of operation”.

In very recent years, mitigations to Return Oriented Programming have been created, falling under the general category of “Control Flow Integrity”, which impose restrictions on the control flow of programs. Techniques involve the detection of changes to the call stack by canaries or via a duplicate ‘shadow’ stack (e.g. in LLVM/Clang), the pre-registration of legitimate entry points to functions (Control Flow Guard in Microsoft Windows), registration of saved return addresses on a ‘call’ (Intel Control-flow Enforcement Technology) and other methods. As is often the case with exploit mitigations, no universally effective method has yet been devised.

To step back for a moment, we have now briefly described several reasons why drawing a clear distinction between code and data is difficult in practice:

- ‘Code’ is stored and manipulated in memory in the same way as ‘data’ (e.g. the Morris Worm finger exploit, where ‘data’ directly becomes ‘code’)
- Parameters to some functions - such as the C standard library function ‘system()’ are essentially ‘code’, rather than data
- Control flow data such as saved return addresses can form Turing-complete languages and should perhaps be considered ‘code’ in its own right
- ‘Code’ exists at multiple levels of abstraction: native ‘machine code’, increasingly higher-level languages which ultimately call native machine code, mechanisms such as shell commands, application macros, and then at the lower levels of abstraction - firmware, microcode, and ultimately the physical hardware.

If we want to control the behaviour of a program, we should probably define “data which controls the behaviour of a program” as code.

This prompts the extremely complex question - what categories of data should we count as “code”?

- Directly executable code?
- Data controlling any Turing-complete behaviour in our program, its dependencies and underlying libraries and technologies?
- Pointers used in control flow?
- Indexes to pointers used in control flow?
- Offsets used in control flow (relative jumps and calls, etc)?
- Any other code pointer, e.g. callbacks, exception handlers, vtable entries, interrupt handlers, handler pointers of any other kind?
- Parameters to any mechanism resulting in arbitrary code execution (e.g. system(), eval(), exec())?
- Parameters to mechanisms which import code into the current address space (dlopen(), LoadLibrary() etc.)?

- Parameters to Just-In-Time compilation mechanisms (all modern web browsers and programs which use web ‘controls’)?
- Arguments used in expressions to calculate indexes or references to pointers used in control flow?
- Indexes to arrays (remember the heap overflow “arbitrary write” idea)?
- Arguments used in expressions to calculate indexes to arrays?
- Sizes of data specified numerically?

Whatever your view of where the line between code and data should be drawn, it should hopefully be clear from this list that there is considerable room for debate.

The Problem, Part 2 - Code Injection

Several categories of security vulnerability are described using the word “injection”; SQL injection, Command Injection, Code Injection, XML injection, template injection and so on. These vulnerabilities typically occur when a “string” variable is created using user-supplied input, and the string is then passed to some other API, interpreter or mechanism for processing. For example, a SQL query might be composed in this way:

```
sQuery = "select * from users where (username ='" +
request.username + "') and password = ('" + request.password + "')"
```

If the user were to supply a username of the form

```
Robert'); DROP TABLE Students;--
```

… the ‘Students’ table would be deleted. The issue here is that the string is code - in this case a SQL query - being dynamically created using user-supplied input. The developer presumably expected the input to be data - in the form of an alphanumeric username and password, perhaps ‘Robert’, and ‘LittleBobby123’, resulting in the SQL query:

```
select * from users where (username ='Robert') and password = ('LittleBobby123')
```

But the user has instead supplied an input containing code, which “breaks out” of the quote-delimited context of the data, and is executed by the database server:

```
select * from users where (username ='Robert'); DROP TABLE
Students;--') and password = ('LittleBobby123')
```

(Bobby Tables is the subject of a notorious XKCD comic: https://xkcd.com/327)

Other code injection vulnerabilities operate in similar ways; command injection relates to composed strings being passed to command shells, “Code Injection” applies to composed strings being passed to dynamic code execution mechanisms - interpreters such as the JavaScript and Python ‘eval()’ functions. In each case, input intended as data is interpreted as code.

With the emergence of Large Language Models, and especially the rise of Agentic AI, this boundary between code and data is even harder to determine and enforce, and the consequences of confusion are perhaps even more impactful.

The Problem, Part 3 - Agentic AI

The most recent incarnation of this problem is that Agentic AI systems typically make no distinction between instructions, in the sense of “things they are asked to do”, and data, in the sense of “input data or the results of tool execution”. This leads to an analogous situation to the arbitrary code execution problem and the code injection problem described above, whereby an attacker can control the Agentic system by submitting instructions in the guise of input data.

Agentic AI Background

Large Language Models (LLMs) have recently emerged as a surprisingly powerful and flexible mechanism to manipulate text. The underlying mechanism involves the application of deep learning (via artificial neural networks) to the task of predicting what the next text token in a sequence of text tokens will be. This simple idea has profound results; it turns out that in order to accurately predict the next token, the mechanism must make deeply-abstracted associations between groups and sequences of tokens, essentially enabling the mechanism to generate text at varying levels of abstraction. Training a model on questions and answers, for instance, will cause the model to generate answers in response to questions - to a varying degree of competence, based on model capacity and training data volume and quality.

The mechanism becomes strikingly powerful when very large neural networks are used, increasing the capacity of the system to store associations, and when the system is trained on very large amounts of relevant text. Modern "frontier" LLMs are trained on extremely large datasets, approaching the scale of all written human language. This results in correlations between patterns of tokens that are both deep and surprisingly subtle. For example, you can ask an LLM to describe a forest glade, and it will output a reasonable description. You can then ask it to make its description more "right wing", for example, and it may change its description to include phrases associated with "right wing" in the context of forests, such as responsible exploitation of natural resources, the benefits of personal independence, living off the land, and so on. Alternatively, if you ask the model to make its description more "left wing", it may introduce phrases emphasising the importance of public ownership of natural resources, public access to green spaces, preservation of habitats, the mental health benefits of touching grass, and so on.

The immense scope of their training data enables LLMs to function as general-purpose manipulators of almost any form of text. They can serve as a limited search engine or a summary tool, they can translate between languages and they are very effective at performing transformations of bodies of text. They are also capable of writing programs in a variety of programing languages and they are able to make use of external tools, when their output is parsed by an agentic application server.

It is this final capability that has led to the development of "Agentic AI" systems; semi-autonomous systems that apply the generative capabilities of Large Language Models to utilise tools, in order to achieve some objective specified in natural language.

In the course of our consulting work for customers, we review a great many AI systems, including - recently - agentic systems. Mechanisms such as Model Context Protocol (MCP) and Agent To Agent (A2A) are being deployed by organisations to enable the creation of rich, flexible, AI-driven services that integrate tightly with existing infrastructure and applications. To get an idea of the use cases for Agentic AI, it's worth reviewing the "Official Integrations" section in the Model Context Protocol github page: https://github.com/modelcontextprotocol/servers

In this section, you can see integrations with major cloud platforms such as Azure, AWS and Google Cloud, databases such as Microsoft SQL Server, PostgreSQL and Oracle, integrations with programming languages, command shells, payment systems such as PayPal and Stripe, and even individual organisations such as the Professional Golfers' Association (PGA). Agentic applications can make use of integrations like these to access services and mechanisms running locally and remotely. This is achieved - in brief - by prompting the LLM with a description of the tools available to it, and the task required, and the LLM responds with an API call it would like an agentic application server (such as an MCP server) to make on its behalf.

The prompt to the LLM is not strictly 'code', but - although the LLM is not directly executing code - the output of the LLM is being interpreted by the agentic server as instructions; the server is running API calls on the LLM's behalf. Depending on the API in question, and the mode of operation of the agentic server, this may literally be actual code (as is the case in smolagents "code agents"), or it may result in the creation or execution of arbitrary code, as is the case in shell or programming language APIs.

Exploiting Agentic AI Systems

One common pattern we are seeing emerge in our consulting work is that *any* data in that appears in the context window of an LLM is capable of profoundly changing the output, via "prompt injection" and other similar techniques. In the context of Agentic systems, this enables attacks such as data extraction - retrieving sensitive data from databases, credentials, personal data of other users and so on, data modification - changing properties of users or systems, executing arbitrary code or commands, and otherwise performing actions more associated with "traditional" vulnerabilities.

Since the interactions that agentic systems carry out - queries, authentications, shell commands - are the direct result of the output of the LLM, the ability to change the output of the LLM carries significant consequences. Compounding this issue is the fact that the LLM can often be induced to facilitate the attack; in an arbitrary code execution attack, the attacker must carefully craft an extremely specific payload, whereas in an Agentic AI attack the attacker can simply reference their objective, and the LLM will create appropriate, helpful content as a continuation of whatever is already in its context window. We have seen examples of LLMs decrypting sensitive data, executing commands, creating administrative users and more, simply because the text in their context window triggered a continuation that resulted in the appropriate APIs being called, with the desired level of privilege.

As concrete examples of this, several vulnerabilities of related types have been publicly disclosed:

CVE-2025-55319, "Agentic AI and Visual Studio Code Remote Code Execution Vulnerability"

This issue is a command injection vulnerability, with a NIST/NVD severity rating of 9.8 out of 10. The issue relates to Agentic integrations with the Visual Studio Code editor. The root cause is that prompt injections in externally-supplied content can cause the agent to perform actions such as modifying sensitive configuration files, ultimately resulting in code and/or command injection.

As you might gather from the discussion of Agentic integrations above, the specific attack vectors related to this "prompt injection" class of vulnerability can vary, but it's important to understand that they aren't limited to malicious source code; any content retrieved by the agent may result in these behaviours; in the context of this general type of software development agent, that may include comments and notes associated with the code, commentary relating to issues, web pages retrieved by the agent, email, messages or chat ingested by the agent, or indeed behaviours created by the prompt or training data of the LLM itself. These issues are characterised as "unauthenticated remote command injection / code execution" for good reason.

Other, similar issues have been disclosed in Claude Code (CVE-2025-54795, a command injection, severity 8.8), Github Copilot (CVE-2025-53773, again, remote command injection via prompt injection, severity 7.8) and several other Agentic development environments and tools. This specific category of issues is one that we are increasingly encountering in our consulting work, in the broader context of Agentic systems.

A second category of Agentic AI security issues relates to "traditional" security vulnerabilities whose attack vectors are increased by their presence within an Agentic framework. A recent case of this is CVE-2025-53967, "Remote Code Execution in Framelink Figma MCP Server", which is a command injection vulnerability within tools exposed by an MCP server. An attacker can connect directly to the MCP server and issue the vulnerable tool calls, but the vulnerable interfaces are also accessible to other components in the Agentic system, since these are tools that the system is making available to LLMs.

Finally, there are several security issues related to the relative recency of these technologies; a good example is CVE-2025-3248, an unauthenticated Remote Code Execution (RCE) issue in Langflow, a popular tool used to build Agentic AI tools. The issue relates to an unauthenticated API endpoint which allows arbitrary python code to be validated (/api/v1/validate/code). By supplying decorators (functions which take functions as arguments and return other functions) or default arguments, arbitrary python code can be executed. Although these are issues within Agentic systems, they fall into the general category of "generic" security vulnerabilities, in that they don't relate directly to instructions emitted by LLMs.

Several projects have been created to mitigate these issues via technology, standards and collaboration, notably

- The Agent To Agent Security Framework, https://www.a2as.org/
- The OWASP Agentic Security Initiative, https://genai.owasp.org/initiatives/agentic-security-initiative

This is an extremely fast-moving area of engineering and research, but initial indications are that - as with buffer overflow exploit mitigations - there may be no universally effective method to mitigate these vulnerabilites.

Conclusions

While the Countesses comments relating to “figures” and “the symbols of operation” relate mainly to mathematical notation, the points she made were strikingly prescient. It is surprising that even in 1843, the author of the first published computer program was aware of the importance of the distinction between code and data, the potential for confusion between the two, and the far-reaching potential of “the symbols of operation”.

In the same set of notes, the Countess commented on “the powers of the Analytical Engine” in a manner that applies to computing machines in general, but also (in the opinion of this author) directly to the present state of artificial intelligence:

(Taken from section “Notes by the translator, Note G”)

It is desirable to guard against the possibility of exaggerated ideas that might arise as to the powers of the Analytical Engine. In considering any new subject, there is frequently a tendency, first, to overrate what we find to be already interesting or remarkable; and, secondly, by a sort of natural reaction, to undervalue the true state of the case, when we do discover that our notions have surpassed those that were really tenable.

The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform. It can follow analysis; but it has no power of anticipating any analytical relations or truths. Its province is to assist us in making available what we are already acquainted with. This it is calculated to effect primarily and chiefly of course, through its executive faculties; but it is likely to exert an indirect and reciprocal influence on science itself in another manner. For, in so distributing and combining the truths and the formulæ of analysis, that they may become most easily and rapidly amenable to the mechanical combinations of the engine, the relations and the nature of many subjects in that science are necessarily thrown into new lights, and more profoundly investigated.

This is a decidedly indirect, and a somewhat speculative, consequence of such an invention. It is however pretty evident, on general principles, that in devising for mathematical truths a new form in which to record and throw themselves out for actual use, views are likely to be induced, which should again react on the more theoretical phase of the subject. There are in all extensions of human power, or additions to human knowledge, various collateral influences, besides the main and primary object attained.

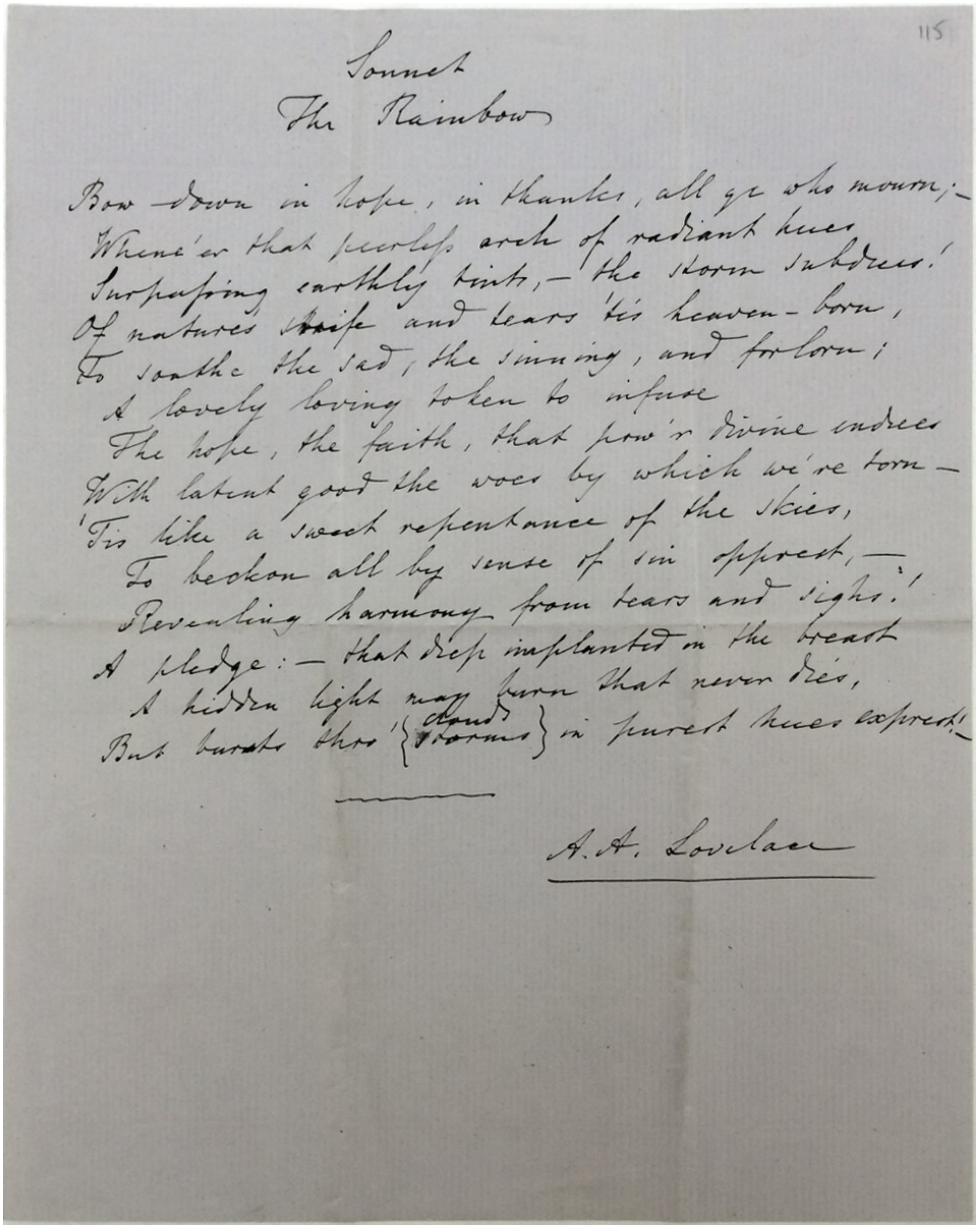
A Final Note

As this article has dealt with some fairly deep, technical and perhaps somewhat depressing issues, it seems appropriate to leave the final words to the Countess herself, in the form of a rather beautiful sonnet she wrote about a rainbow:

The Rainbow

Bow down in hope, in thanks, all ye who mourn;—
Where'in that peerless arche of radiant hues
Surpassing earthly tints,—the storm subdues!
Of nature's strife and tears 'tis heaven-born,
To soothe the sad, the sinning, and forlorn;
A lovely loving token to infuse
The hope, the faith, that pow'r divine endures
With latent good the woes by which we're torn.—
'Tis like a sweet repentance of the skies,
To beckon all by sense of sin opprest,—
Revealing harmony from tears and sighs!
A pledge:—that deep implanted in the breast
A hidden light may burn that never dies,
But bursts thro' clouds in purest hues exprest!

A. A. Lovelace



115

Sonnet
The Rainbow

Bow down in hope, in thanks, all ye who mourn;—
Where'er that peerless arch of radiant hues,
Surpassing earthly tints,— the storm subdues!
Of natures strife and tears 'tis heaven-born,
To soothe the sad, the sinning, and forlorn;
A lovely loving token to infuse
The hope, the faith, that pow'r divine endues
With latent good the woes by which we're torn—
'Tis like a sweet repentance of the skies,
To beckon all by sense of sin opprest,—
Revealing harmony from tears and sighs!
A pledge:— that deep implanted in the breast
A hidden light may burn that never dies,
But bursts thro' {clouds storms} in purest hues exprest!—

A.A. Lovelace

(Image courtesy Wikimedia / Somerville College, Oxford)

Acknowledgements

I'd like to thank my colleagues Andy Davis, Liz James and Thomas Atkinson for their help in producing this article, as well as members of the NCC BNRD Research Group - including but not limited to Jonathan Granewich, Andrew Havard, Thomas Atkinson (again), Ben Williams, Daniel Wilkinson, Nathan Hays, Sean Spratt, Andrew Wade, Rob Russell, Jose Selvi, Curtis Fletcher, David Brauchler, Emily Liu, Ristin Rivera, Julian Yates, Jacques Ernotte and Tomasz Chrosniak for their patience, help and support in many discussions around the security of what we should probably, reluctantly now be calling 'AI'.