The use of Static Analysis to detect Malware in Embedded Systems

C Sampson, J G Drever, B Third, Atkins UK – The Barbican, East Street, Farnham, Surrey, GU9 7TB

Keywords: Malware, Embedded Systems, Static Analysis

Abstract

The use of Formal Static Analysis techniques to ensure that software-based safety systems are free from compiler introduced errors is well established (Pavey, Winsborrow, 1995) [1]. This technique ensures that the executable binary code created by the compiler is mathematically equivalent to the original source code. This paper reports on extending this technique to detect malware inserted into executable code. The Source-Code Comparison process was originally developed by British Energy for the verification of the Primary Reactor Protection System software of the Sizewell 'B' Nuclear Power Plant. The process takes the executable binary file that is resident on the target computer and re-creates the equivalent assembler code using disassembler tools. This is then formally compared to the original source code using the MALPAS Compliance Analysis tool, and any discrepancies are revealed. The process has the ability to detect any executable binary code that cannot be traced back to the source code, and may therefore be used to detect the presence of malware in the executable. The paper will report on experiments conducted by Atkins to determine whether malware that has been deliberately inserted into the executable can be detected using Formal Proof. The applicability of the process to software developed for general purpose operating systems (e.g. Windows) will also be evaluated.

1 Introduction

Many of our most trusted devices, critical to our everyday life and wellbeing, contain embedded software. From our cars, to our phones, heating and cooling, lighting, domestic appliances, and televisions all contain embedded microprocessors running software. Further, our national infrastructure: communications systems, water supply, food production, electricity and gas, petrol pumps, road traffic signalling and trains – things we rely on to keep our economy moving - all rely on software. We also rely on embedded software to keep us safe – for example in nuclear reactors, medical devices, avionics & air traffic control, police and defence systems. Embedded software is prolific and vital to our everyday lives.

Malicious hackers use malware to:

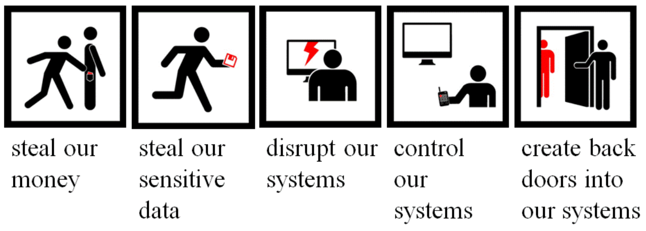


Figure 1: Malware attacks

Traditional Malware detection and prevention relies on two key techniques:

1. Detecting when the embedded software is doing something that it probably shouldn't be doing (eg transmitting data, accessing information)

2. Detecting patterns of software known as being Malware.

Preliminary results from Symantec published in 2008 suggested that "the release rate of malicious code and other unwanted programs may be exceeding that of legitimate software applications."[2] According to F-Secure, "As much malware [was] produced in 2007 as in the previous 20 years altogether [3]. Malware is prolific and threatens our everyday lives.

This paper presents another detection and prevention method using a technique previously employed to detect errors in compilers – **formal proof that embedded software is mathematically identical to the source code that generated it.**

2 How does Malware enter embedded systems and what do we currently do about it?

2.1 The software lifecycle

Malware attacks on our desktops and laptops are well understood ; backdoor code, Trojan horses, worms and viruses are all well-known techniques. However attacks on embedded software are more complex, many embedded applications are not open to the internet and communicate only across segregated and tightly controlled networks.

To understand the vulnerabilities, it is worth looking at a typical embedded software development lifecycle.

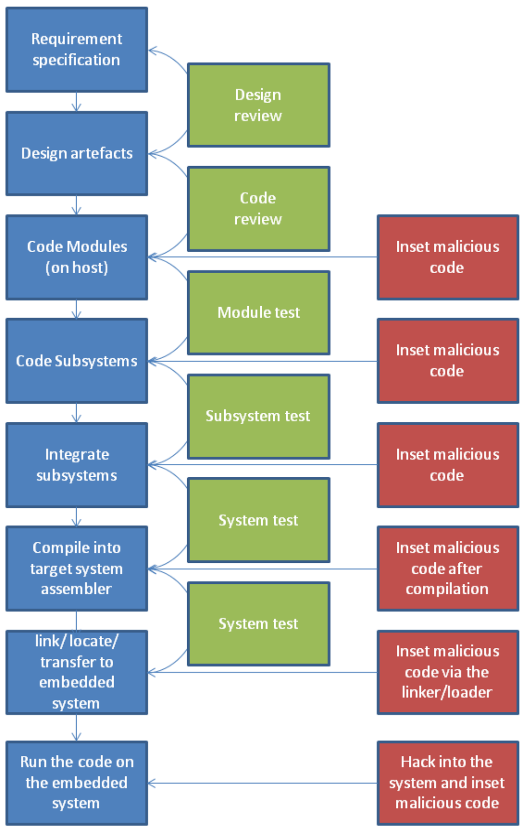

Requirement specification
Design artefacts
Code Modules (on host)
Code Subsystems
Integrate subsystems
Compile into target system assembler
link/locate/ transfer to embedded system
Run the code on the embedded system
Design review
Code review
Module test
Subsystem test
System test
System test
Inset malicious code
Inset malicious code
Inset malicious code
Inset malicious code after compilation
Inset malicious code via the linker/loader
Hack into the system and inset malicious code

Figure 2: Standard software lifecycle and vulnerabilities

Here are a few methods of inserting Malware into an embedded system:

1. **Develop malicious code (or pay a developer to write some code) during the development lifecycle.** This could be mitigated during a standard peer review process or independent test, but there is a residual risk that peer review and module test could miss something very subtle.

2. **Create malicious code and hide it from other developers – then include the malicious code at compilation time.** Independent testing might find this, however most testing is based on positive affirmation that functions are correctly coded. There might be a possibility of finding this code if there is MCDC and full path coverage at assembler level.

3. **Include malicious code at target cross-compilation time.** System testing might find this if you are very lucky. If it is common Malware, then Malware detection could pick up (if you run the Malware scanner on the target code!). If you have exceptional defensive code, you might detect something strange happening during run time.

4. **Develop code which modifies your code during patches**. Patches can be verified using hashing algorithms and comparing the hash output to a fixed length signature. However, there is a risk that patches are not always checked for integrity.

5. **Include malicious code while the binary file is being transferred to the target hardware.** This technique is very hard to detect and can have a high impact if successful.

6. **Write code which modifies your code during run-time.** Real-time analys is would be required for this to be detected. However, the real-time analysis could affect the system and in itself compromise system integrity.

2.2 Current methods for detecting malware

Currently detecting embedded malware is a complex process involving advanced forensic techniques and cryptographic hashing functions. The main issues surrounding these techniques are the high cost of maintenance and the technical issues which can affect the control systems which they are designed to protect.

There are three main techniques of detecting malware:

1. **Anomaly based detection**: after the detection engine has learned what forms a clean/safe environment, the detection engine can then detect malware which forces the system outside normal operation. The drawback of this type of detection is the high false alarm rate.

2. **Specification based detection**: in order to reduce the high false alarm rate of anomaly based detection, specification based detection attempts to approximate the requirements of a system as opposed to the behaviour of a system.

3. **Signature based detection:** signature detection uses known attack patterns to detect attacks on a system. This is a good approach, however, if an unknown attack is launched against a system then it is unlikely to be detected as the signature has not been added to the detection database.

Each technique has its uses, mainly in a corporate environment. The main issue with each of the techniques is the level in which they interact with the system. These techniques are great at detecting network based attacks however the systems have to be taken off line in order to check for malware on each device. This approach may be unsatisfactory if the system has long periods of run-time and little maintenance down-time.

In addition to the limitations to detection, it is still possible that the malware has been designed to evade the malware detection engine by learning how the system operates and adapting accordingly. This is a technique used by some of the newest pieces of malware available.

2.2 Current methods for detecting malware

Currently detecting embedded malware is a complex process involving advanced forensic techniques and cryptographic hashing functions. The main issues surrounding these techniques are the high cost of maintenance and the technical issues which can affect the control systems which they are designed to protect.

There are three main techniques of detecting malware:

1. Anomaly based detection: after the detection engine has learned what forms a clean/safe environment, the detection engine can then detect malware which forces the system outside normal operation. The drawback of this type of detection is the high false alarm rate.

2. Specification based detection: in order to reduce the high false alarm rate of anomaly based detection, specification based detection attempts to approximate the requirements of a system as opposed to the behaviour of a system.

3. Signature based detection: signature detections uses known attack patterns to detect attacks on a system. This is a good approach, however, if an unknown attack is launched against a system then it is unlikely to be detected as the signature has not been added to the detection database.

Each technique has its uses, mainly in a corporate environment. The main issue with each of the techniques is the level in which they interact with the system. These techniques are great at detecting network based attacks however the systems have to be taken off line in order to check for malware on each device. This approach may be unsatisfactory if the system has long periods of run-time and little maintenance down-time.

In addition to the limitations to detection, it is still possible that the malware has been designed to evade the malware detection engine by learning how the system operates and adapting accordingly. This is a technique used by some of the newest pieces of malware available.

3 Formal Proof that the Executable is the same as the Code

3.1 Introduction

During the development of the Sizewell B nuclear power station, a software verification method was devised to identify any defects introduced into the Reactor Protection System software due to compiler errors. There was a worry that the compilation process used to generate the code for the target system may include bugs that might cause safety issues. The process devised made use of the MALPAS toolset to mathematically prove that the binary image on the embedded system was identical to the high level language code (PLM 86). The vital part of MALPAS that is unique in this context is the ability to mix 3rd generation language and assembler in a single mathematical model through the use of the MALPAS Intermediate Language (IL).

Further to this, Atkins recently performed a feasibility study for EDF and Areva that proved a similar for Teleperm XS using the C language and Intel Assembler.

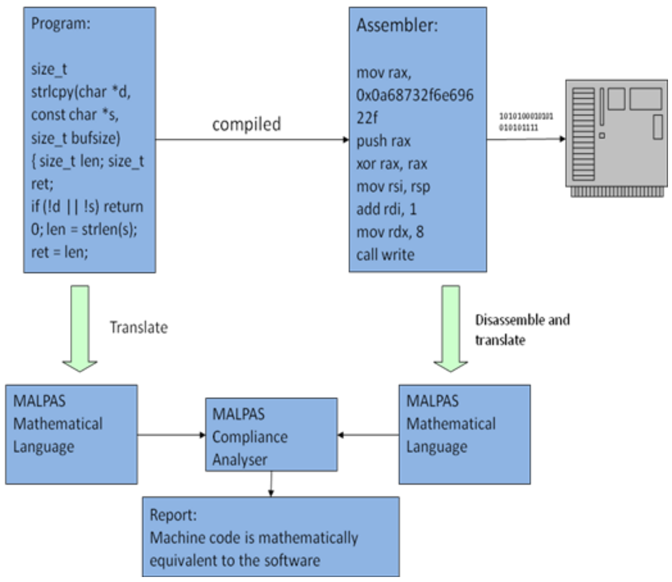
From a **safety** perspective we were looking to prove:

1. **Program state.** The process checks that all writes to memory and the stack are semantically consistent between the source code and target code.
2. **Constant initialisation.** Note that as a special case of item 1, the process will need to check that any initial values loaded directly into memory by the compiler are consistent between the target and source.
3. **Procedure calls.** The process checks that the address of a procedure called in the ASM and the equivalent C function call in the source are consistent.
4. **Stack consistency.** The process should check that the value of the stack pointer at the end of the procedure is equal to that at the start of the procedure.
5. **No added code**. The process should ensure that no code has been added to the executable that is not traceable to the C source code.

From a security perspective we would want to check #5 – there is No added code.

Process to Check for No Added Code

The process for checking that there is no added code is simplified below.


Program:
compiled
Assembler:
1010100010101
010101111
Translate
Disassemble and translate
MALPAS Mathematical Language
MALPAS Compliance Analyser
MALPAS Mathematical Language
Report: Machine code is mathematically equivalent to the software

```
size_t
strlcpy(char *d,
const char *s,
size_t bufsize)
{ size_t len; size_t
ret;
if (!d || !s) return
0; len = strlen(s);
ret = len;
```

```
mov rax,
0x0a68732f6e696
22f
push rax
xor rax, rax
mov rsi, rsp
add rdi, 1
mov rdx, 8
call write
```

Figure 3: Simplified process for proving "no added code"

In some more detail, we follow the following steps:

Step 1: Disassemble and translate

1.1 Convert the binary file on the target into Hexadecimal

1.2 Convert the Hexadecimal File into Assembler

1.3 Translate the Assembler into MALPAS Mathematical Language (IL)

Step 2: Translate the Source code (in this case "C") into IL

Step 3: Perform MALPAS compliance analysis

3.1 Convert the C IL into malpas_check proof assertions and insert those into the correct places in the ASM IL.

3.2 Run the MALPAS compliance analyser on the ASM IL.

3.3 Check the output for discrepancies (there might be compiler errors as well as Malware).

Step 4: Report

4 Experimentation with MALPAS

4.1 Method of Experiment

The example below is of a single C statement (Figure 3). The Assembler produced by the compiler is given (Figure 4), and finally the IL Translation of the assembler (Figure 5). Note that the code assertion is given by the "malpas_check()" statement. It is necessary to replace the variable references in the C Translation by stack accesses (as this is how the variables are located by the compiler).

```
Sum = param1.v + param2.v + param3.v;
```

Figure 4: C Source code

```
                                 @2:
006F  C45E08          LES      BX,[BP+args]
0072  26C47704        LES      SI,ES:[BX+4]
0076  26D904          FLD      ES:[SI]
0079  C45E08          LES      BX,[BP+args]
007C  26C47708        LES      SI,ES:[BX+8]
0080  26D804          FADD     ES:[SI]
0083  C45E08          LES      BX,[BP+args]
0086  26C45F0C        LES      BX,ES:[BX+12]
008A  26D807          FADD     ES:[BX]
008D  DD56F2          FST      [BP+Sum]
```

Figure 5 – Intel Assembler from disassembled executable image

```
 00220:      ; [ @2: ]
 00230:     MAP [ LES     BX,[BP+ARGS] ]
                 bx := stack !! (bp+args);
                 es := stack !! (bp+args + 2);
            ENDMAP;
            bl := low(bx);
            bh := high(bx);

00240:      MAP [ LES     SI,ES:[BX+4] ]
               si := mem !! adr(es, bx+4);
               es := mem !! adr(es, bx+4 + 2);
            ENDMAP;

00250:      [ FLD     ES:[SI] ]
            fsp := fsp + 1;
            fstack := update (fstack,
                                 fsp,
              real_value (mem!!!adr(es, si)));

00260:      MAP [ LES     BX,[BP+ARGS] ]
                  bx := stack !! (bp+args);
                  es := stack !! (bp+args + 2);
            ENDMAP;
            bl := low(bx);
            bh := high(bx);

00270:      MAP [ LES     SI,ES:[BX+8] ]
              si := mem !! adr(es, bx+8);
              es := mem !! adr(es, bx+8 + 2);
            ENDMAP;

00280:      [ FADD    ES:[SI] ]
            fstack := update (fstack, fsp,
fstack ! (fsp) + real_value (mem!!!adr(es,
si)));

00290:      MAP [ LES     BX,[BP+ARGS] ]
               bx := stack !! (bp+args);
               es := stack !! (bp+args + 2);
            ENDMAP;
```

```
           bl  := low(bx);
           bh  := high(bx);

00300:     MAP [ LES     BX,ES:[BX+12] ]
              bx  := mem !! adr(es, bx+12);
              es  := mem !! adr(es, bx+12 + 2);
           ENDMAP;
           bl  := low(bx);
           bh  := high(bx);

    00310: [ FADD    ES:[BX] ]
           fstack := update (fstack, fsp,
fstack ! (fsp) + real_value (mem!!adr(es,
bx)));

    00320: [ FST     [BP+SUM] ]
           stack := updated(stack, bp+dlsum,
store_real (fstack ! fsp));
```

[ASSERTION to check that the C functionality is preserved in the Assembler]
[Sum is replaced with real_value(stack!!!(bp+dlsum)), derived from the memory map produced by the compiler

Note that most of the time C is dealing with the objects whereas ASM is dealing with pointers to the objects which are located in mem or stack. The C objects are therefore replaced by the equivalent stack operations.
]

```
$malpas_check ((real_value(stack!!!(bp+sum))

  EQ

real_value(deref_dword_pointer($cmem,
deref_dword_pointer($cmem,
$cstack!!!(bp+args),4),0))

+real_value(deref_dword_pointer($cmem,
deref_dword_pointer($cmem,
$cstack!!!(bp+args),8),0))

+real_value(deref_dword_pointer($cmem,
deref_dword_pointer($cmem,
$cstack!!!(bp+args),12),0)));
```

Figure 6 – IL Translation of Assembler, with proof assertion shown in red.

A compliance analysis result of threat := false; indicates that the Assertion has been proved, and no malware has been inserted.

Further examples will be developed to demonstrate that malware inserted into the executable results is a threat at the Compliance Analysis stage.

5 Use in Practice

This process can be applied to any translatable embedded software. We would envisage that once the target image is analysed once, then it could be saved and checked against the executable image on a regular basis to prove that no changes have been made. This could be performed using a simple checksum utility.

The initial process should only take 2-3 weeks to execute once the tools have been created to disassemble the code and translate the assembler.

Types of Vulnerable System

We believe that this process would be useful for the following applications:

- Secure communications systems including crypto
- Industrial control systems (Civil Nuclear, Utilities, High Value Manufacturing)
- High BIL systems (cooling, life support systems)
- Weapon systems
- Air traffic control and avionics
- Financial transaction systems
- EPOS
- Automotive systems
- Robotics

References

[1] Formal demonstration of equivalence of source code and PROM contents, Proceedings of the IMA Conference on Mathematics of Dependable Systems, Oxford University Press, 1995, pp225-248 D J Pavey and L A Winsborrow.

[2] Symantec Internet Security Threat Report: Trends for July–December 2007 (Executive Summary) XIII. Symantec Corp. April 2008. p. 29.

[3] F-Secure Quarterly Security Wrap-up for the first quarter of 2008. F-Secure. March 31, 2008.