European Commission
Directorate-General Home Affairs

Prevention, Preparedness and Consequence Management of Terrorism and other Security-related Risks Programme

HOME/2009/CIPS/AG/C2-050
i-Code: Real-time Malicious Code Identification

Deliverable D2: System Implementation

Workpackage:	WP2: System Implementation
Contractual delivery date:	December 2011
Actual delivery date:	December 2011
Deliverable Dissemination Level:	Public
Editor	Paolo Milani (TUV), Martina Lindorfer (TUV)
Contributors	FORTH, POLIMI, EURECOM, VU
Internal Reviewers:	VU, POLIMI

Executive Summary: In this deliverable, we describe the implementation of the i-Code real-time malicious code detection system, focusing on its subsystems and their integration. The system brings together the components of network-level attack vector detection forensics tools, techniques for the classification and clustering of shellcode based on the control flow graph, and behavioral-based malware detection by benign-malicious action comparisons. It also incorporates techniques for malcode detection in high-speed networks.

With the support of the Prevention, Preparedness and Consequence Management of Terrorism and other Security-related Risks Programme. European Commission - Directorate-General Home Affairs[†].

[†]This project has been funded with the support of the Prevention, Preparedness and Consequence Management of Terrorism and other Security-related Risks Programme of European Commission - Directorate-General Home Affairs. This publication reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Contents

List of Figures

CHAPTER 1

Introduction

The i-Code project aims to detect and analyze malicious code and Internet attacks in real time. Its scope includes the detection of attacks in the network and on the host, the analysis of the malicious code, and post-attack forensics. Thus, the project takes on challenges from different aspects of operational security and proposes to address them with a number of novel detection and analysis tools. Due to the variety of the tasks they face, these tools are very diverse, but they can be interconnected to provide an enriched understanding of security incidents, by means of the i-Code console.

In the previous deliverable *D1: System Design*, we discussed the design of each i-Code detection and analysis component. Furthermore, we explored the potential synergies between these components and provided an early blueprint for the i-Code console. In this document, we continue from there, and report on the implementation of the i-Code components that has been carried out by the project partners.

Outline In this document, we describe the implementation of the i-Code tools, highlighting all the individual components, as well as the way in which they will be integrated. In Chapter 2, we discuss our techniques for shellcode detection based on network level emulation. In Chapter 3, we present our system for performing behavioral analysis of the collected shellcodes, and to classify them based on the structure of their (unpacked) code. Chapter 4 describes our techniques for detecting malware on the end host by contrasting its behavior against normal behavior patterns obtained from real, uncompromised machines. Chapter 5 describes the scalable, high/performance I/O architecture that we developed to speed up payload execution. Chapter 6 provides more detail on the i-Code console, including not just the web-based frontend, but also the backend architecture and data model. Finally, in Chapter 7 we lay out our plans for deploying the i-Code tools to a testbed to verify their functionality and effectiveness.

CHAPTER 2

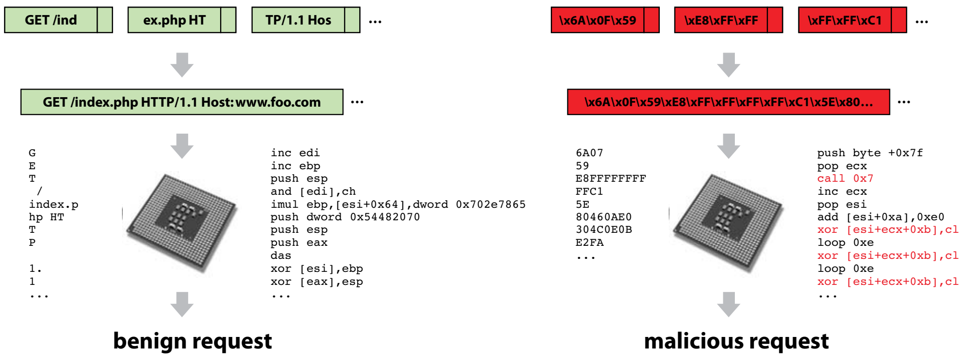
Network-level Emulation

Network-level emulation is an effective approach for the detection of code injection attacks, based on the identification of the shellcode that is part of the attack vector. In this chapter we describe the implementation of Nemu, our research prototype network-level attack detector used in i-Code. Although Nemu aims to detect server and client side attacks at the network level, its core detection engine is generic and can identify the presence of shellcode in arbitrary inputs.

2.1 Scanning Network Traffic

Emulation-based detection aims to identify the mere presence of shellcode in arbitrary data streams. The principle behind this approach is that, due to the high density of the IA-32 instruction set in terms of available opcodes, any piece of data can be interpreted as valid machine code and can be treated as such. For example, in the same way an input that is treated as a series of bytes can be inspected using string signatures or regular expressions, when the same input is interpreted as a series of machine instructions, it can then be examined using code analysis techniques like static code analysis or code emulation.

The machine code interpretation of arbitrary data results to random code which, when it is attempted to run on an actual CPU, usually crashes soon, e.g., due to the execution of an illegal instruction. In contrast, if some input contains actual shellcode, then this code will run normally, exhibiting a potentially detectable behavior, as illustrated in Figure 2.1. Shellcode is nothing more than a series of assembly instructions, usually crafted as position-independent code that can be injected and run from an arbitrary



```
G
E
T
 /
index.p
hp HT
T
P

1.
1
...
```

```
inc edi
inc ebp
push esp
and [edi],ch
imul ebp,[esi+0x64],dword 0x702e7865
push dword 0x54482070
push esp
push eax
das
xor [esi],ebp
xor [eax],esp
...
```

```
6A07
59
E8FFFFFFFF
FFC1
5E
80460AE0
304C0E0B
E2FA
...
```

```
push byte +0x7f
pop ecx
call 0x7
inc ecx
pop esi
add [esi+0xa],0xe0
xor [esi+ecx+0xb],cl
loop 0xe
xor [esi+ecx+0xb],cl
loop 0xe
xor [esi+ecx+0xb],cl
...
```

Figure 2.1: Overview of network-level emulation. After TCP stream reassembly, each network request is interpreted as machine code and is loaded on a CPU emulator. The execution of the random code corresponding to a benign request usually ends abruptly after a few instructions, while the execution of an actual shellcode exhibits certain detectable behaviors.

location in a vulnerable process, and thus its execution can be easily simulated using merely a CPU emulator.

Nemu passively captures network packets using `libpcap` [13], reassembles TCP/IP streams using `libnids` [19], and then scans the client-initiated part of each TCP connection using the runtime heuristics presented in the previous deliverable *D1: System Design*. The default input buffer size is set to 128KB, which is large enough for typical service requests. Especially for web traffic, pipelined HTTP/1.1 requests through persistent connections are split into separate streams. Otherwise, an attacker could evade detection by filling the stream with benign requests until exceeding the buffer size.

2.2 Shellcode Execution

Our goal is to detect network streams that belong to code injection attacks by passively monitoring the incoming network traffic and identifying the presence of shellcode. Each request to some network service hosted in the protected network is treated as a potential attack vector. The detector attempts to execute each incoming request on a CPU emulator as if it were executable code.

Instruction set emulation has been implemented interpretively, with a typical fetch, decode, and execute cycle. Accurate instruction decoding, which is crucial for the identification of invalid instructions, is performed using `libdasm` [9]. For our prototype, we have implemented a subset of the IA-32 instruction set, including most of the general-purpose instructions, but no FPU, MMX, SSE, or SSE2 instructions, except `fstenv/fnstenv`, `fsave/fnsave`, and `rdtsc`. However, *all* instructions are fully decoded, and if during execution an unimplemented instruction is encountered, the emulator proceeds normally to the next instruction.

The implemented subset suffices for the complete and correct execution of the decryption part of all the shellcode implementations that we used during our testing. Even highly obfuscated shellcode generated by the TAPiON polymorphic shellcode engine [5], which intersperses FPU instructions among the decoder code, is executed correctly, since FPU instructions are used only as NOPs and do not take part in the useful computations of the decoder.

Since the exact location of the shellcode in the input data is not known in advance, the emulator repeats the execution multiple times, starting from each and every position of the stream. In certain cases, however, the execution of some code paths can be skipped to optimize runtime performance [17].

2.3 Detection Heuristics

Our approach for the generic detection of previously unknown shellcode is based on runtime detection heuristics that match inherent execution patterns found in different shellcode types. All heuristics are evaluated in parallel and are orthogonal to each other, which means that more than one heuristic can match during the execution of some shellcode, giving increased detection confidence. For example, some heuristics match the decryption process of polymorphic shellcodes, while others match operations found in plain shellcode. Since any polymorphic shellcode carries an encrypted version of a plain shellcode, the execution of a polymorphic shellcode usually triggers both self-decrypting and plain shellcode heuristics.

The overall concept can be thought as analogous to the operation of a typical signature-based intrusion detection system, with some key differences: each input is treated as code instead of a series of bytes, the detection engine uses code emulation instead of string or regular expression matching, and each "signature" describes a generic, inherent behavior found in all instances of a particular type of malicious code, instead of an exploit or vulnerability-specific attack vector.

The execution of self-decrypting shellcode is identified by two key runtime behavioral characteristics: the execution of some form of *GetPC* code, and the occurrence of several *self references*, i.e., read operations from the memory addresses of the input stream itself, as illustrated in Fig 2.2. The GetPC code is used by the shellcode for finding the absolute address of the injected code, which is mandatory for subsequently decrypting the encrypted payload, and involves the execution of an instruction from the `call` or `fstenv` instruction groups [16].

Besides the GetPC self-decrypting shellcode detection heuristic [16], the rest of the heuristics used in Nemu are mostly based on memory accesses

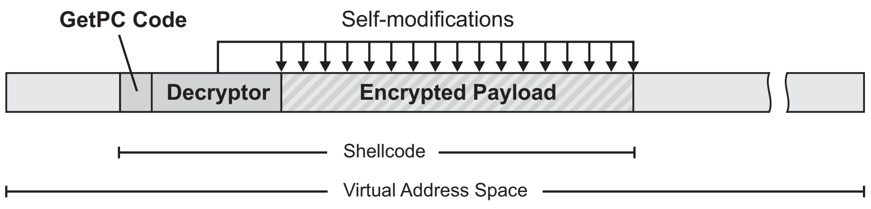


Figure 2.2: A typical execution of a polymorphic shellcode using network-level emulation. Its self-decrypting behavior is identified by two key runtime characteristics: the execution of some form of *GetPC* code, and the occurrence of several *self references*.

to certain locations in the address space of a vulnerable Windows process. To emulate the execution of these accesses correctly, the virtual memory of the emulator is initialized with an image of the complete address space of a typical Windows XP process taken from a real system. The image consists of 971 pages (4KB each), including the stack, heap, PEB/TIB, and loaded modules. All four heuristics use the same memory image and thus can be evaluated in parallel during execution (the same holds for the fifth GetPC heuristic, which does not rely on the memory image at all).

Among other initializations before the beginning of a new execution [16], the segment register `FS` is set to the segment selector corresponding to the base address of the Thread Information Block, the stack pointer is set accordingly, while any changes to the original process image from the previous execution are reverted.

The runtime evaluation of the heuristics requires keeping some state about the occurrence of instructions with an operand that involved the `FS` register, as well as about read and write accesses to the memory locations specified in the heuristics. Note that for conditions that specify a pointer access, as for example the reading of the `PEB.LoaderData` pointer in condition P2, all four bytes of the pointer should be accessed. If during some execution the address of `PEB.LoaderData` is read by an instruction with a byte memory operand, condition P2 will hold only when all three remaining bytes of the pointer are also read by subsequent instructions. This kind of pedantic checks enhance the robustness of the heuristics by ruling out random code that would otherwise accidentally match some of the conditions.

Regarding the SEH-based memory scanning heuristic (described in *D1: System Design*), although SEH chain validation is more complex compared to other instrumentation operations, it is triggered only if conditions S1 and S2 are true, which in practice happens very rarely. If upon the execution of some instruction S1 and S2 are satisfied but S3 is not, then SEH chain validation is performed after every subsequent instruction that performs a memory write.

When an `int 0x2e` instruction is executed, the `eax` register is checked for a value corresponding to one of the system calls that can be used for memory scanning. Although the actual functionality of the system call is not emulated, the proper return value is stored in the `eax` register depending on the validity of the supplied memory address. In case of an egg-hunt shellcode, this behavior allows the scanning loop to continue normally, resulting to several system call invocations.

CHAPTER 3

Shellcode Analysis and Classification

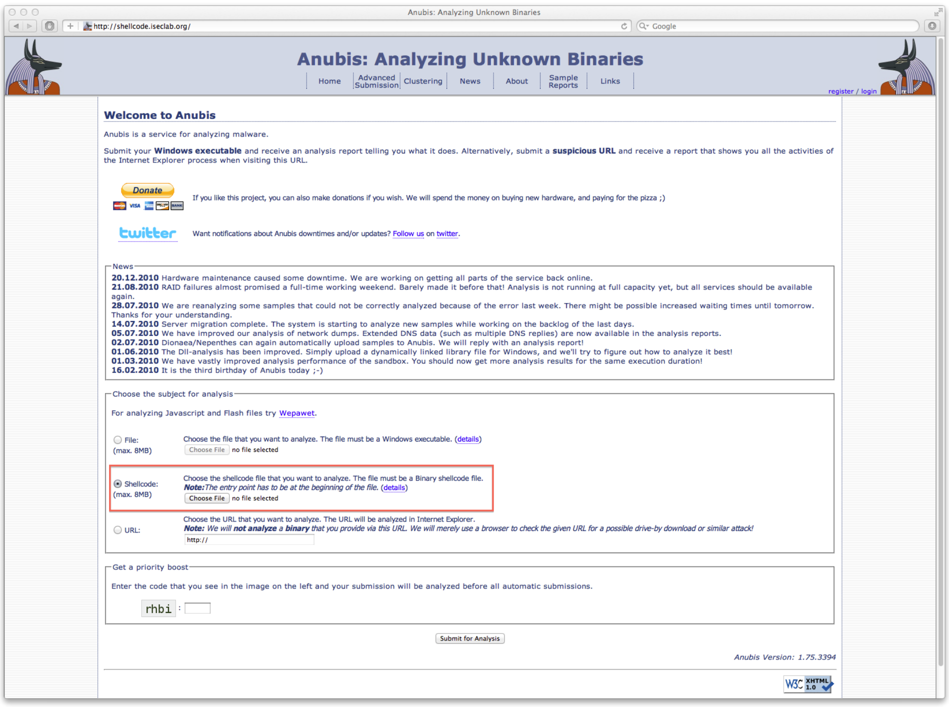
Our aim is to analyze shellcodes that are being used by attackers in the wild in order to study their variety and phylogeny. In this chapter we describe our infrastructure for collection shellcode samples as well as our dynamic analysis tool for analyzing a shellcode behavior and obtaining unpacked code. Based on the unpacked version of a shellcode's code we then use static analysis techniques and perform unsupervised clustering of the collected shellcodes.

We based the implementation of the dynamic shellcode analysis on the Anubis malware analysis tool [7, 4, 6], a dynamic malware analysis system based on an instrumented Qemu [8] emulator. It is offered as an open service through a public website, where users can submit binaries for analysis, and receive a report that describes the behavior of the sample in a human-readable way.

3.1 Shellcode Collection

In order to integrate the submission of shellcodes from various sources we extended the web interface of Anubis (illustrated in Figure 3.1). This interface is publicly available at `http://shellcode.iseclab.org/` and can be used by anybody anonymously to submit samples. Registered users can use this interface to submit shellcodes analyzed with a higher priority than anonymous submissions.

Apart from manual submissions to the web interface we also offer Python scripts for the automated submission of shellcodes. We use these scripts for the submission of shellcodes detected by Nemu [17] as well as batch submissions from external partners.



Anubis: Analyzing Unknown Binaries

Welcome to Anubis

Anubis is a service for analyzing malware.

Submit your **Windows executable** and receive an analysis report telling you what it does. Alternatively, submit a **suspicious URL** and receive a report that shows you all the activities of the Internet Explorer process when visiting this URL.

If you like this project, you can also make donations if you wish. We will spend the money on buying new hardware, and paying for the pizza ;)

Want notifications about Anubis downtimes and/or updates? Follow us on twitter.

News

20.12.2010 Hardware maintenance caused some downtime. We are working on getting all parts of the service back online.
21.08.2010 RAID failures almost promised a full-time working weekend. Barely made it before that! Analysis is not running at full capacity yet, but all services should be available again.
28.07.2010 We are reanalyzing some samples that could not be correctly analyzed because of the error last week. There might be possible increased waiting times until tomorrow. Thanks for your understanding.
14.07.2010 Server migration complete. The system is starting to analyze new samples while working on the backlog of the last days.
05.07.2010 We have improved our analysis of network dumps. Extended DNS data (such as multiple DNS replies) are now available in the analysis reports.
02.07.2010 Dionaea/Nepenthes can again automatically upload samples to Anubis. We will reply with an analysis report!
01.06.2010 The Dll-analysis has been improved. Simply upload a dynamically linked library file for Windows, and we'll try to figure out how to analyze it best!
01.03.2010 We have vastly improved analysis performance of the sandbox. You should now get more analysis results for the same execution duration!
16.02.2010 It is the third birthday of Anubis today ;-)

Choose the subject for analysis

For analyzing Javascript and Flash files try Wepawet.

File: (max. 8MB) — Choose the file that you want to analyze. The file must be a Windows executable. (details) no file selected

[✓ Shellcode:] (max. 8MB) — Choose the shellcode file that you want to analyze. The file must be a Binary shellcode file. ***Note:****The entry point has to be at the beginning of the file.* (details) no file selected

URL: — Choose the URL that you want to analyze. The URL will be analyzed in Internet Explorer. ***Note:*** *We will* ***not analyze*** *a* ***binary*** *that you provide via this URL. We will merely use a browser to check the given URL for a possible drive-by download or similar attack!* http://

Get a priority boost

Enter the code that you see in the image on the left and your submission will be analyzed before all automatic submissions.

rhbi :

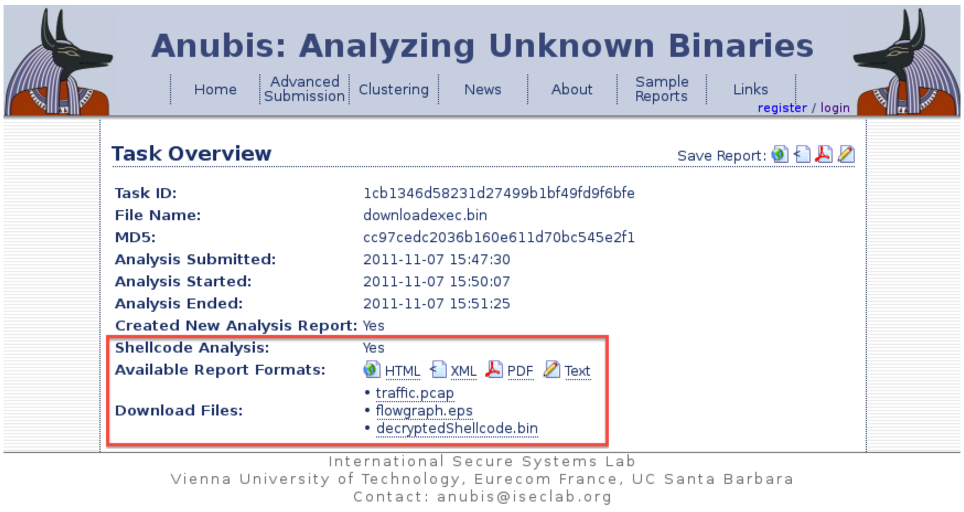
Anubis Version: 1.75.3394

Figure 3.1: Interface for submitting shellcode samples to dynamic analysis.

3.2 Shellcode Analysis

Shellcode samples are submitted as binary blobs. Thus, as a first step we need to transform the shellcode in an executable format analyzable by Anubis. This enables us to perform a high-level analysis of the submitted shellcodes and to monitor behavior such as downloading files, or opening ports on the victims computer. To this end we wrap the shellcode into a Windows PE executable template that consists of the header of a PE executable with its entry point set to the end of the file, so that when we append the shellcode to the end of this template it gets executed when the executable is run. Optionally we can also set the entry point to some other location inside the shellcode.

Anubis analyzes the Windows PE executable template containing a shellcode sample in an emulated environment and logs its Windows API and Windows Native API calls. After the shellcode terminates or a timeout has been reached, Anubis provides a detailed report of the shellcode's high-level behavior in various formats (HTML, XML, PDF and text), as well as a dump of its network activity (*traffic.pcap*) produced with libpcap [13]. We extended this existing reporting functionality (see Figure 3.2) to provide two additional analysis artifacts: A control flow graph (*flowgraph.eps*) and the unpacked and decrypted version of the shellcode (*decryptedShellcode.bin*).



Task ID:	1cb1346d58231d27499b1bf49fd9f6bfe
File Name:	downloadexec.bin
MD5:	cc97cedc2036b160e611d70bc545e2f1
Analysis Submitted:	2011-11-07 15:47:30
Analysis Started:	2011-11-07 15:50:07
Analysis Ended:	2011-11-07 15:51:25
Created New Analysis Report:	Yes
Shellcode Analysis:	Yes
Available Report Formats:	HTML XML PDF Text
Download Files:	• traffic.pcap • flowgraph.eps • decryptedShellcode.bin

International Secure Systems Lab
Vienna University of Technology, Eurecom France, UC Santa Barbara
Contact: anubis@iseclab.org

Figure 3.2: Interface for accessing the report of the dynamic analysis result for a shellcode sample.

In order to generate these more low-level analysis artifacts we extended Anubis to allow for a more fine-grained analysis of the shellcode itself. To this end, we adapted and integrated the generation of the following log files from previous work [10]:

FlowLog:
Contains the address and size of each executed translated basic block.

SliceLog:
Contains the identification information and contents of each translated basic block.

MemLog:
Stores current instruction pointer and target address of each memory access.

MemDump:
Holds the values that are read or written in the corresponding MemLog entry. If read, this log file has to be iterated parallel to the MemLog file.

For the logging of the FlowLog and the SliceLog we extended Anubis in the following way: Every translated basic block is handed to a callback function in Anubis before it is going to be executed. Therefore by an extension of this function, we log every executed basic block in the FlowLog and its corresponding data, namely the executed code itself, in the *SliceLog*. We also apply filtering of the executed basic blocks so that we only log the user-specific portion of the shellcode and not the execution of system level API calls. In order to generate memory dumps we implemented a new function in Qemu's code translator, which is responsible for translating the code of the guest operating system into executable code on the host operating system. Every time the code translator handles an operation that reads or writes memory we log this operation in the *MemLog* and *MemDump*.

We process the obtained information from these log files to generate the CFG, which represents disassembled translated basic blocks as nodes and transfers in control flow as edges between these nodes. For our implementation we use Graphviz and the DOT language to represent our flow graph, but an extension to different output formats is possible.

For our purposes we adjusted the representation of data in the CFG with three heuristics: First, we simplified the CFG by collapsing nodes that are only connected together by one execution path. Second, we added information about the execution order of API function calls to the nodes in the CFG. Third, we extended the CFG to include information about the existence of decryption layers. We do this by checking if an executed region of code has been written before by another operation in the code (*write then execute heuristic*). Whenever we detect a new layer of decryption during graph generation, we can also dump the current memory of the code. Since during analysis not every execution path will be taken, this allows for a later analysis of the code to check, if there is some interesting information inside the decrypted section.

Figure 3.4 shows a sample CFG generated during the analysis phase. Everything inside the red box corresponds to one decryption layer. That means, this code was decrypted by an operation in the layer above. The number in the curly braces corresponds to the hexadecimal address of the code that actually wrote this layer. Grey nodes represent nodes that include calls to an API function. Figure 3.3 shows one of these nodes, which in that case calls the API function `WinExec`. The label 2 of the API function means that this function was the third API function to be executed during execution.

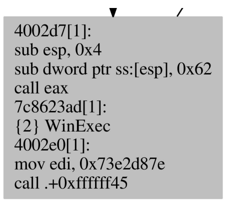
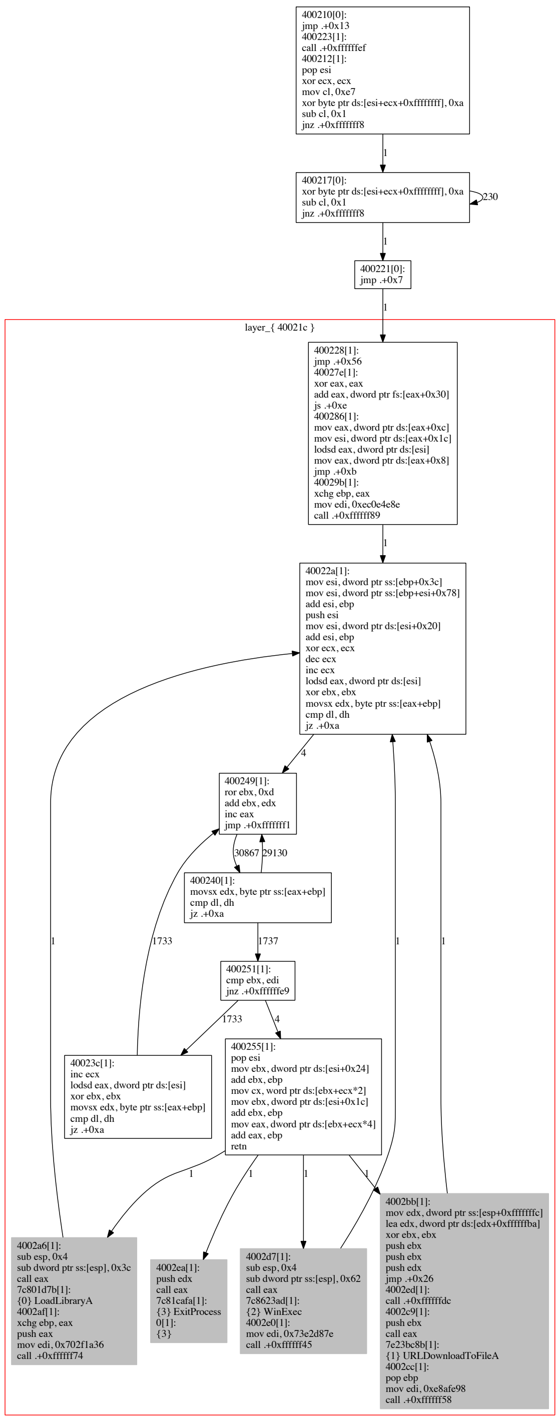

```
4002d7[1]:
sub esp, 0x4
sub dword ptr ss:[esp], 0x62
call eax
7c8623ad[1]:
{2} WinExec
4002e0[1]:
mov edi, 0x73e2d87e
call .+0xffffff45
```

Figure 3.3: Sample node of the control flow graph calling an API function.



```
400210[0]:
jmp .+0x13
400223[1]:
call .+0xffffffef
400212[1]:
pop esi
xor ecx, ecx
mov cl, 0xe7
xor byte ptr ds:[esi+ecx+0xffffffff], 0xa
sub cl, 0x1
jnz .+0xfffffff8
```

```
400217[0]:
xor byte ptr ds:[esi+ecx+0xffffffff], 0xa
sub cl, 0x1
jnz .+0xfffffff8
```

```
400221[0]:
jmp .+0x7
```

```
400228[1]:
jmp .+0x56
40027e[1]:
xor eax, eax
add eax, dword ptr fs:[eax+0x30]
js .+0xe
400286[1]:
mov eax, dword ptr ds:[eax+0xc]
mov esi, dword ptr ds:[eax+0x1c]
lodsd eax, dword ptr ds:[esi]
mov eax, dword ptr ds:[eax+0x8]
jmp .+0xb
40029b[1]:
xchg ebp, eax
mov edi, 0xec0e4e8e
call .+0xffffff89
```

```
40022a[1]:
mov esi, dword ptr ss:[ebp+0x3c]
mov esi, dword ptr ss:[ebp+esi+0x78]
add esi, ebp
push esi
mov esi, dword ptr ds:[esi+0x20]
add esi, ebp
xor ecx, ecx
dec ecx
inc ecx
lodsd eax, dword ptr ds:[esi]
xor ebx, ebx
movsx edx, byte ptr ss:[eax+ebp]
cmp dl, dh
jz .+0xa
```

```
400249[1]:
ror ebx, 0xd
add ebx, edx
inc eax
jmp .+0xfffffff1
```

```
400240[1]:
movsx edx, byte ptr ss:[eax+ebp]
cmp dl, dh
jz .+0xa
```

```
400251[1]:
cmp ebx, edi
jnz .+0xffffffe9
```

```
40023c[1]:
inc ecx
lodsd eax, dword ptr ds:[esi]
xor ebx, ebx
movsx edx, byte ptr ss:[eax+ebp]
cmp dl, dh
jz .+0xa
```

```
400255[1]:
pop esi
mov ebx, dword ptr ds:[esi+0x24]
add ebx, ebp
mov cx, word ptr ds:[ebx+ecx*2]
mov ebx, dword ptr ds:[esi+0x1c]
add ebx, ebp
mov eax, dword ptr ds:[ebx+ecx*4]
add eax, ebp
retn
```

```
4002a6[1]:
sub esp, 0x4
sub dword ptr ss:[esp], 0x3c
call eax
7c801d7b[1]:
{0} LoadLibraryA
4002af[1]:
xchg ebp, eax
push eax
mov edi, 0x702f1a36
call .+0xffffff74
```

```
4002ea[1]:
push edx
call eax
7c81cafa[1]:
{3} ExitProcess
0[1]:
{3}
```

```
4002d7[1]:
sub esp, 0x4
sub dword ptr ss:[esp], 0x62
call eax
7c8623ad[1]:
{2} WinExec
4002e0[1]:
mov edi, 0x73e2d87e
call .+0xffffff45
```

```
4002bb[1]:
mov edx, dword ptr ss:[esp+0xfffffffc]
lea edx, dword ptr ds:[edx+0xffffffba]
xor ebx, ebx
push ebx
push ebx
push edx
jmp .+0x26
4002ed[1]:
call .+0xffffffdc
4002c9[1]:
push ebx
call eax
7e23bc8b[1]:
{1} URLDownloadToFileA
4002cc[1]:
pop ebp
mov edi, 0xe8afe98
call .+0xffffff58
```

Figure 3.4: Sample control flow graph produced during dynamic analysis.

3.3 Shellcode Classification

By clustering the collected shellcodes we can determine how similar they are and whether attackers make large use of exploitation tools (e.g., metasploit) or they write their own custom shellcodes. Based on these clusters we can also classify newly detected shellcodes as either belonging to an existing cluster, i.e. resembling code that we have seen before, or being new code. Therefore, we perform single-linkage agglomerative hierarchical clustering on all of our collected shellcode samples. Single-linkage means that we compute the distance between two clusters as the distance between the two closest elements in the two clusters. Agglomerative means that we merge the clusters iteratively so that our algorithm performs the following steps:

1. Assign each shellcode to a separate cluster so that for N shellcodes we have N clusters each containing only one shellcode. The distances between clusters are the same as the distances between the shellcodes they contain.
2. Find and merge the closest pair of clusters and merge them into a single cluster.
3. Compute the distance between the new cluster and each of the existing clusters.
4. Repeat steps 2 and 3 until all shellcodes are clustered into a single cluster of size N (or until we have reached the desired degree of well-separateness between the clusters).

Instead of implementing this clustering algorithm from scratch we take advantage of existing implementations in the R programming language [3]. Primary candidates for hierarchical clustering in R are the standard R function *hclust* or *agnes* from the cluster package [1]. Another variant of hierarchical clustering is implemented by *fastcluster* [2]. The main benefit of *fastcluster* is its faster and memory-saving implementation which leads to a worst-case time complexity of $\mathcal{O}(N^2)$ instead of $\mathcal{O}(N^3)$ as for *hclust* and *agnes* [15].

The input for the clustering algorithm is a distance (or "dissimilarity") matrix that contains distances between all pairs of shellcodes we collected. For the distance calculation we use the following functions, that operate on the unpacked version of a shellcode's code:

Exedit Distance: The exedit distance was defined by Ma et al. [12] as the relative edit distance over the executed bytes of the shellcode. The executed bytes of the shellcode are concatenated in the order they appear in the payload in order to construct a string representation of the shellcode. The distance between two such strings is calculated as the number of edit operations (insertion, deletion and substitution) that are needed to transform one string into another and normalized over the length of the longer string.

Ma et al. concluded that this distance metric is better than the relative edit distance over the entire shellcode because changes in embedded string constants may greatly influence th metric, if data is not distinguished from code.

CFG Fingerprint Distance: We refined the techniques of Kruegel et al. [11] to calculate the structural distance between two shellcodes. For this we first disassemble the unpacked shellcode and build its CFG from which we then derive fingerprints for k-subgraphs of basic blocks. We represent a shellcode as a set of fingerprints and set the distance between two shellcodes to zero if there is one or more elements in the set intersection of fingerprints. Conversely, we set the distance to one if two shellcodes have no common fingerprints.

CFG Basic Block Distance: Based on the CFG of the unpacked shellcode we calculate another structural distance from the basic blocks of two shellcodes. For this we calculate the ratio of the number of mismatched basic blocks to the number of common basic blocks (the Jaccard distance).

CHAPTER 4

Behavior-based Detection of Malcode

In the previous deliverable *D1: System Design* we described the data set of system call traces that we collected in the wild from ten real-world users. In the rest of this chapter we focus instead on the presentation of the experiments results.

4.1 Detection Results

In this section, we evaluate the effectiveness of a detector based on access activity models. Similar to the analysis for the n-gram model, we ran ten experiments. More precisely, for each experiment, we picked one of the machines. We then used the system call data recorded on the other nine hosts to generate the access activity model, as described in the previous section. Finally, we used this model for detection. For this, we first checked the resource accesses performed by all processes on the machine that was *not* used for model generation. Then, we examined the accesses performed by the malware samples. For each experiment, we evaluated the detection capabilities and false positives of the file system model alone, the registry model alone, and both models combined.

File system access activity model. On average, the file system access activity model contains about 100 labels. These labels contain tokens that restrict read access to about 70 directories, write access to about 80 directories, and execute access to about 30 directories. The results for the file system model are shown in Table 4.1 and Table 4.2. In these tables, we see a number of different columns for the detection rates and the false positive rates. These are discussed in the following paragraphs.

Machine	Detection rate	False positive rate
1	0.656	0.225
2	0.657	0.173
3	0.657	0.154
4	0.657	0.156
5	0.657	0.143
6	0.635	0.242
7	0.657	0.267
8	0.657	0.045
9	0.657	0.025
10	0.657	0.050
Average	0.655	0.148

Table 4.1: Summary of the detection based on our file-system access activity model.

When using the original model to check all read, write, and execute accesses, we see an average detection rate of almost 66% for the malware samples (column *Detection rate*) and a false positive rate of roughly 15% (column *False positive rate*). Note that, similar to the experiments with the n-gram models, the false positive rates are computed on the basis of applications and not processes.

At first glance, the results appear sobering. However, a closer examination of the result reveals interesting insights. First, we decided to investigate the false negative rate in more detail. When looking at the execution traces of the malware programs, we observed that many samples did not get far in their execution but quickly exited or crashed. Interestingly, a substantial fraction of malicious samples never wrote to the file system or the registry, and they did not open any network connections. It is difficult to confirm that these samples exhibit any malicious activity at all. In fact, this calls into question the occasionally very high detection rates of the n-gram-based model, and it further confirms our previous insight that system call sequences are not closely related to actual malicious behavior. As a result, we decided to remove all samples that never perform a write operation or open a network connection (or socket) from our malware data sets. This decreases our malware data set to 7,847 samples that exhibit at least some kind of activity. It also improves our detection rate to more than 90%, as can be seen in column *Adjusted detection rate* of Table 4.2. For the remainder of this paper, all reported detection rates are computed based on the adjusted malware data set.

In the next step, we investigated the false positives in more detail. Table 4.2 shows the access violations for each machine, divided into violations due to read (column *Read*), write (column *Write*), and execute (column *Execute*) access attempts. It can be seen that execute violations account for

Machine	*Adjusted detection rate*	*Access violations rate*			*Detection rate (only writes)*	*Final detection*	
		Read	*Write*	*Execute*		*FP rate*	*Det. rate*
1	0.906	0.000	0.022	0.222	0.864	0.0	0.864
2	0.907	0.000	0.011	0.172	0.902	0.0	0.902
3	0.907	0.000	0.130	0.043	0.902	0.0	0.902
4	0.907	0.024	0.049	0.122	0.902	0.0	0.902
5	0.907	0.024	0.024	0.095	0.902	0.0	0.902
6	0.877	0.014	0.055	0.242	0.868	0.0	0.868
7	0.907	0.020	0.041	0.265	0.901	0.0	0.901
8	0.907	0.000	0.045	0.000	0.902	0.0	0.902
9	0.907	0.000	0.025	0.000	0.902	0.0	0.902
10	0.907	0.000	0.038	0.038	0.902	0.0	0.902
Average	0.904	0.008	0.044	0.137	0.895	0.0	0.895

Table 4.2: Detection based on our file-system access activity model (details).

a significant majority of false positives. However, we also found that they are only marginally important for detection. Thus, for the next experiment, we decided to use only the access tokens that refer to `write` operations. This is justified by the fact that we are most interested in preserving the integrity of the operating system resources. The detection results for the new *write-only* detection approach are presented in column *Detection rate (only writes)* of Table 4.2. As can be seen, the numbers remain high with 89.5%. This confirms that write access violations are a good indicator for malicious activity. With this approach, the false positives are identical to the write violations, which are shown in column *Write*.

We further examined the reasons for the remaining write violations. It turned out that these violations were due to two root causes. One set of false positives was caused by our own system-call logging component that wrote temporary files directly into the `C:\` directory before sending the data over the network. The second reason was due to software updates. More precisely, we detected a number of cases in which an application was writing to its folder in `C:\Program Files`. Of course, only this program had read/execute access to that directory. However, we never saw a write access during training, and as a result, the directory was considered read-only. To accommodate for updates, we manually added a rule to the model that would grant write permission to applications that "own" directories in `C:\Program Files`. Moreover, we granted our component write access to `C:`. With more extensive training, both access activities would have very likely been added automatically. The model that incorporated our minor adjustments generated no more false positives, as shown in column *Final detection/FP rate*. However, the detection capabilities of the model remain

Machine	*Detection rate*	*False positive rate*	*Det. rate (only writes)*	*FP rate (only writes)*	*Final det. rate*
1	0.567	0.063	0.530	0.063	0.521
2	0.557	0.107	0.540	0.053	0.521
3	0.566	0.179	0.530	0.128	0.062
4	0.557	0.000	0.530	0.000	0.540
5	0.557	0.000	0.530	0.000	0.540
6	0.557	0.015	0.530	0.000	0.540
7	0.597	0.133	0.530	0.000	0.540
8	0.557	0.067	0.530	0.067	0.537
9	0.561	0.100	0.530	0.025	0.521
10	0.557	0.000	0.530	0.000	0.540
Average	0.563	0.066	0.530	0.034	0.486

Table 4.3: Detection based on our registry access activity model.

basically unchanged, as shown in column *Final detection/det. rate.*

Registry access activity model. On average, the registry access activity model contains about 3,000 labels, significantly more than the file-system model. The labels contain tokens that restrict read access to about 1,600 keys and write access to about 2,800 keys (execute is not defined for registry keys).

The results for the registry model are shown in Table 4.3. The columns *Detection rate* and *False positive rate* show the detection rates and the false positive rate, respectively, for the original model. It can been seen that both the detection rate and the false positive rates are lower than for the file system model. We also examined the detection rate and the false positive rate when considering only write operations (columns *Det. rate (only writes)* and *FP rate (only writes)*). Similar to the file system case, the false positive rate drops significantly; there are five runs in which no false positives were reported at all. However, the detection rate remains (relatively) high.

We also examined the cases for which the registry access model raises false positives. We found that all registry write access violations can be attributed to the sub-tree `HKEY_USERS\Software\Microsoft`. While this is an important part of the registry that contains a number of security settings, we wanted to understand the detection capabilities of a model that permits write access to these keys. To this end, we added a manual rule to allow writes to this sub-tree and re-run the experiments on the malware data set. We see that the model is still effective and achieves a detection rate of over 48% (shown in column *Final det. rate* of 4.3) with no false positives. Considering the significantly larger size of the registry models compared to the ones for the file system, we expect that a larger training set would be required to effectively capture legitimate writes to the `Software\Microsoft` sub-tree.

Full access activity model. For the final experiment, we combined those improved file system and registry models that yielded zero false positives. The combined detection rate improves compared to the file system model alone, but only slightly (between 1% and 2% for all of the ten runs). The average detection improved from 89.5% to 91% (of course, with no false positives).

4.2 Discussion

When focusing on write operations only, our access activity model achieves a good detection rate (more than 90%) with a very low false positive rate. The false positive rate even drops to zero with minor manual adjustments that compensate for deficiencies in the training data, while still retaining its detection capabilities. This suggests that a system-centric approach is suitable for distinguishing between benign and malicious activities, and it handles applications not previously seen well. This happens because most benign applications are written to be good operating system “citizens” that access and manage resources (files and registry entries) in the way that they are supposed to. In fact, as we can observe from our results, out of 242 distinct applications seen in our experiments, policy violations occurred only for few, specific classes of programs (system utilities, AV software). On the other hand, violations of n-gram models, occurred across the board.

Malicious programs frequently violate good behavior, often because their goals inevitably necessitate tampering with system binaries, application programs, and registry settings. Of course, we cannot expect to detect all possible types of malicious activity. In particular, our detection approach will fail to identify malware programs that ignore other applications and the OS (e.g., the malware does not attempt to hide its presence or to gain control of the OS) and that carry out malicious operations only over the network. For these types of malicious code, it will be necessary to include also network-related policies. Finally, the data collection overhead is very low and enforcing the generated models is even faster, since no writes (for logging) occur.

CHAPTER 5

A Scaleable I/O Architecture

In deliverable *D1: System Design*, we discussed an architecture for high-speed network processing. In this chapter, we will flesh out the architecture and discuss the implementation.

An *I/O architecture* is the communication fabric linking applications, OS kernel and hardware, that extends from library interfaces in userspace down to peripheral devices. It crosscuts the classical OS layering. The present I/O architecture not only impedes performance on conventional hardware, it also obstructs the use of heterogeneous hardware. As the bottleneck in system software has moved from the CPU to the memory system (especially for I/O-intensive tasks such as networking), we need an I/O architecture that alleviates pressure on the memory subsystem.

As mentioned in D1, the i-Code I/O architecture avoids common bottlenecks by reconfiguring datapath logic at application load time to match workload and exploit special-purpose hardware. In general, the two extremes in coping with heterogeneous hardware and software configurations are (a) not to deal with it at all (static code), and (b) fully recompile all the code with specific optimizations for each specific configuration. We find a mid-way point between static code and full recompilation, because both are impractical. The first cannot anticipate all computer architectures and applications. The second requires a single tool-chain capable of programming all available devices on each end host.

Instead, we construct application-tailored I/O paths at runtime from sets of precompiled processing and buffering elements. It avoids bottlenecks by optimizing the mapping of applications onto the physical computer architecture: a tailoring algorithm selects the set of elements that (1) satisfies the application, (2) maximizes the use of specialized hardware and (3) minimizes data movement, in that order.

The two components of the I/O architecture that are crucial for performance are processing and buffering. For processing, we reuse the well-known streams and filters model and for buffering, we opt for a buffer management system where all live data is kept in coarse-grain ring buffers, and buffers are shared long-term between protection domains. Moreover, actual data transformation is replaced with updates to metadata structures, wherever possible.

In summary, we developed an I/O architecture with the following properties:

1. a buffer management system for I/O that avoids common copy, context switch and cache miss overhead through shared memory transport and indirection.

2. a dataplane that bypasses bottlenecks and integrates all hardware by selecting suitable implementations of logic on-demand to form I/O paths: graphs of processing and buffering elements.

3. a control system that automatically translates application requests expressed as abstract Unix-like pipelines into I/O path implementations tailored to the application profile and local hardware characteristics. On top of this we have engineered legacy I/O interfaces to allow direct comparison with Linux.

When we evaluated the architecture we found that it:

I Increases Unix primitive throughput 2x to 30x over standard Linux.

II Increases legacy application throughput up to 4x.

III Increases throughput further when modified call semantics are allowed.

IV Enables intrinsically efficient and portable native applications.

5.1 Buffering

A buffer management system (BMS) controls the movement of data through a computer system. Its design determines the number of copies, data-cache misses and task switches per block. To maximize end-to-end throughput, these technical buffering details must be managed centrally, concealed from individual data clients, such as applications and filters.

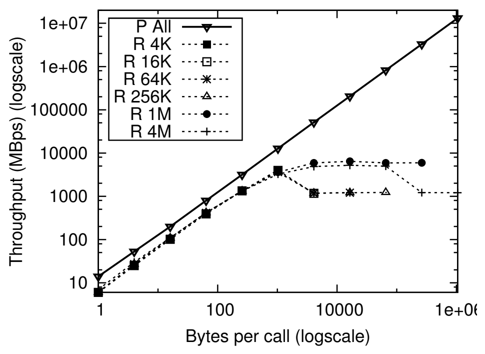


Figure 5.1: Copy avoidance with `peek`. For clarity, peek ('P') results are collapsed, because they all overlap.

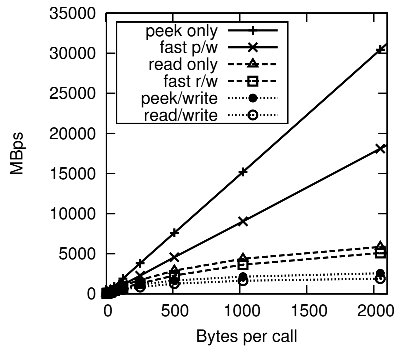


Figure 5.2: Throughput with splicing ("fast")

5.1.1 Peek

One performance drawback of Unix I/O is that it implements expensive copy semantics, that is, `read` and `write` create private copies of blocks for the caller. To avoid this cost, we extend the API with `peek(int, char **, int)`, a `read`-like function that uses weak move semantics. With `peek`, a client receives a direct pointer into the stream. The `read` call, then, is nothing more than a wrapper around `peek`, `memcpy` and exception handling logic. Figure 5.1 shows the gains obtained by switching from a copy-based `read` (R) to an indirect `peek` (P) call. The figure plots throughput for various DBuf sizes at increasing call sizes (size of the application buffer passed in `read` and `write` calls). As expected, peek throughput scales linearly for all buffers, as it is purely computational. Read throughput, on the other hand, experiences `memcpy` overhead. Even for the smallest packets, it is about one third slower than peek.

5.1.2 Ring buffers

Static, shared rings hold a number of advantages over I/O based on dynamically allocated blocks: they amortize allocation and virtual memory management operations over the lifetime of streams and render sequential access cheap within streams because blocks are ordered in memory. As mentioned in D1, we use shared ring buffers for data and indices.

The transport system that we use supports multiple buffers. Interestingly, we can actually exploit this requirement by specializing buffer *implementations* to fit the task profile or hardware at hand. For example, packet reception rings allow overflow, while IPC rings implement blocking semantics.

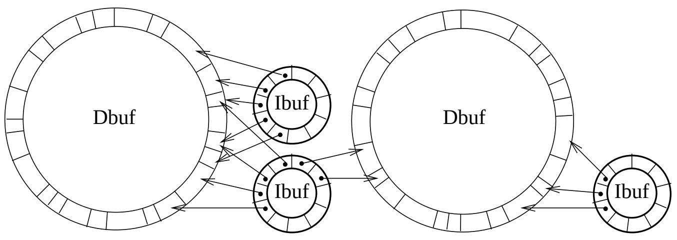


Figure 5.3: Chain of data rings holding data and index buffers holding pointers to data

Device driver buffers match the hardware specification of their specific device. In the end, we weave this array of buffers together into a coherent transport system with indirect buffers, to which we now focus our attention.

5.1.3 Indirection

Presenting clients with an idealized view of private, sequential streams conflicts with copy-avoidance through buffer sharing. A shared packet reception ring holds multiple entangled application streams. When one client needs write access to a shared resource it effectively asks for an independent stream. We can disentangle streams through copying, but that is expensive. The alternative is to use a type of indirection, such as hardware protected virtual memory.

We replace pointers with indices and pointer queues with index buffers or "IBufs", that store the indices. Figure 5.3 shows the interoperation of IBufs with DBufs. IBufs differ from pointers in two ways: they replace direct addressing with globally valid lookup structures and add a small set of metadata fields.

The main feature of indices is their lookup structure: a "rich" pointer. Indices must be able to address buffer contents across virtual memory protection domains. Each index implements a three-level lookup structure consisting of a systemwide unique identifier of a DBuf, a block index within this buffer, and an optional offset plus length pair to select a region within the block (e.g., a TCP segment within an Ethernet frame).

Indices from different IBufs may share access to the same DBuf and indices within the same IBuf may point to blocks in multiple DBufs. Figure 5.3 shows both situations. The first situation is common when multiple clients need a private view on data in a shared ring, which we discussed before. The second situation occurs when a client needs to access multiple rings, e.g., a server listening on two NICs.

Resolving rich pointers is more expensive than following regular pointers, but this cost is amortized by caching a translation for subsequent accesses within the same space. Handling "buffer-faults" is more costly. Such exceptions occur when a referenced DBuf is not mapped into the current memory protection domain. To maintain the illusion of globally shared memory, buffer-faults are handled similar to demand paging: an index pointing to a buffer that is not accessible causes the kernel to map the buffer in the task's virtual memory (after verifying access permission).

5.1.3.1 Transparent indirection

The BMS shields clients from indirection details: IBufs present the same file interface as DBufs and perform read- and write-through to referenced DBufs internally, so that clients can remain unaware of which they are accessing. Reading from an IBuf entails resolving the rich pointer and then calling the `peek` method of Section 5.1.1 of the mentioned DBuf. Writing to an IBuf also involves selecting a DBuf as backing store; currently each space appoints one default buffer. Such transparent indirection enables copy avoidance behind the interface, known as *splicing* [14].

5.1.3.2 Splicing

When a write request to an IBuf involves data that already resides in a DBuf, write-through can be avoided. This situation occurs often, not in the least because we incorporate the disk cache as a DBuf. On top of IBufs we have implemented splicing: generic, copy-free data transfer between streams.

Results We now quantify the effects of splicing on throughput for both `peek` and `read`, whereby we do not optimize the read call through page access revocation. Figure 5.2 shows the relative efficiency in transferring data from a DBuf to IBuf, by plotting each method's throughput against call size. The test is indicative of file servers, for instance, where data is read from the page cache and written to the network transmission buffer. The fastest mechanism is *peek only*: the `peek` equivalent of read-only access. This mechanism processes even faster than the physical bus permits, because no data is touched. The method serves no purpose; we only show it to set an upper bound on the performance. About half as fast is *fast peek/write*, which combines `peek` with splicing. This, too, does not actually touch any data, but writes out an IBuf element. Overhead caused by `read` can be seen by comparing these two results with those of *read only* and *fast read/write*. They are 3x slower still. Worst results are obtained when we cannot use splicing, but instead must write out data: throughput drops again, by another factor 2.5. This experiment clearly shows that combined gains from copy avoidance are almost an order of magnitude (9x) when all data is cached. Savings will be even higher for buffers that exceed L2, because then the large blocks will cause many more d-Cache and TLB misses than the small IBuf elements.

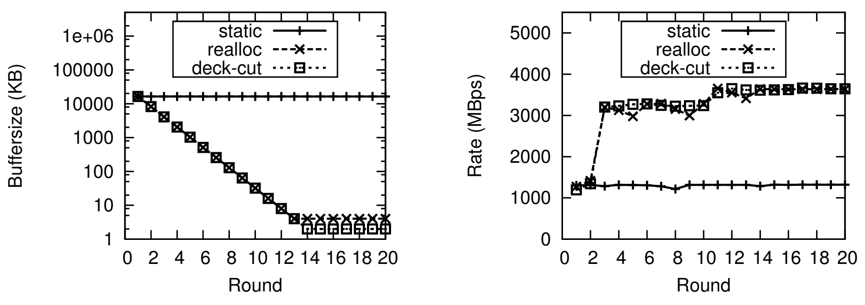
5.1.4 Size

The size of a buffer influences its maximum throughput in two ways: larger buffers reduce synchronization overhead (such as task-switching), but smaller buffers experience fewer cache misses. We therefore make our buffers variable size, where the size is adjusted at runtime. We call such buffers 'self-scaling'. They adapt their size at runtime based on "buffer pressure": the distance between producer and consumer, normalized to buffer size. If pressure goes above a high-water mark a ring grows; if it drops below the opposite, it shrinks. We have implemented two types of scaling: 'reallocation' and 'deck-cut'. Both are handled behind the interface, i.e., transparent to the user.

Reallocation replaces one memory region with another of a different size. A reallocation operation can only be started without copying when the producer reaches the end of the region (i.e., when it would otherwise wrap around). As long as consumers are accessing the old region, both regions must be kept in memory. The approach is similar to rehashing and has the same drawback: during reallocation the buffer takes up more space than before. Deck-cut avoids this problem. It allocates a maximum-sized buffer, but can temporarily disable parts of it, in a manner similar to how a deck of cards is cut: everything behind the cut is left unused. Deck-cut is computationally cheaper than reallocation, because the only required action is to move the pointer indicating the start of the ring. As a result, it is well-suited to highly variable conditions. We exploit this characteristic by moving the watermarks closer together. A drawback is that it never returns memory to the general allocator.

Scaling is restricted by a few technical considerations. In our I/O architecture, indices (including the read and write pointers) must be monotonically increasing numbers (i.e., they are not reset during a wrap), because those tell in which loop through the buffer – and in the case of reallocation in which memory region – an index falls. To learn the offset of a block in a memory region, one calculates the modulo of the number of slots in the ring (S). When a buffer scales, S changes. To guarantee correct offset calculation for all sizes, modulo operations must always overlap. In other words, all values of S must be natural multiples of the same base. The higher the base, the faster the buffer expands and contracts (we only use base 2).

Results Figure 5.4 compares copy (i.e., `write` followed by `read`) throughput for a static buffer of 16MB with that of rings that gradually self-scale down from 16MB until they stabilize. A round denotes a decision moment where the buffer can scale: a moment when the producer wraps around. Figure (a) shows that both scaling buffers continue to decrease buffer size at each opportunity. Figure (b) shows that, instead of scaling linearly with buffer size, throughput sees three levels that correspond with access from



(a) Buffer size (b) Throughput

Figure 5.4: Effects of scaling at runtime, when each round cuts the buffer size in half

main memory, L2 and L1 caches, respectively. The increase in throughput between main memory and L2 is significant: a three-fold improvement. Deck-cut scales further down than reallocation as a result of the moved watermarks.

5.2 Processing and signaling

The second main component concerns processing. We construct I/O applications on-demand from processing and buffering elements. Such *application tailored-I/O* has the potential to maximize throughput, by moving logic close to data and by exploiting unique hardware features.

Streams are point-to-point channels between filters. They map trivially onto ring buffers, to form a store and forward network where each data block is saved at each edge. But, the reading and writing of indices – let alone data blocks – exceeds the operational cost of many I/O operations. We avoid this unnecessary storage cost in the common case. If two connected filters execute in the same memory protection domain and no external parties (such as controlling application logic) require on-demand access to the interconnecting stream, it schedules the second filter immediately after the first and passes a pointer in-memory rather than saving an index (or the original payload) to an intermediate buffer.

Merging filter execution at runtime in this manner increases data cache hitrate and removes non-functional scheduling and memory access overhead. The optimization can be applied recursively throughout a request graph. In practice, we optimize away nearly all ring buffer accesses, because memory protection crossings are few and application logic is commonly interested only in a single stream of the I/O path: the end-result of all transformations.

Passing data blocks between execution spaces (e.g., from the kernel to userspace) is expensive in traditional I/O architectures. Function calling is not an option. Here we must resort to another, more expensive, method.

Polling and interrupt driven processing are standard approaches, but both have drawbacks: polling wastes cycles at low rates and interferes with scheduling; interrupts incur cost at high rates, exactly when the system is already stressed. Hybrid systems, such as interrupt moderation, clocked interrupts, or interrupt masking (as implemented in Linux NAPI), evade both pathological cases. We combine interrupt moderation with timeouts to amortize cost at high rates while bounding worst case delivery latency. Its approach is unique in that both the interrupt moderation threshold and timeout value can be set individually for each buffer. This way, the trade-off between efficiency and latency can be tuned to application constraints.

Results Figure 5.5 shows pipe throughput offset against call size at various levels of interrupt moderation factor. All results are obtained with a large DBuf of 16MB, or 8000 slots, so that we have a wide measurement range for signal moderation. We send at least one signal per timeout epoch (in the experiment set at 1000HZ) which limits moderation benefit for very high numbers. The figure shows that, indeed, throughput scales below linear. When batching up to 32 signals we already achieve 92% of the maximally obtainable gain (with factor 2048): 2.36x versus 2.56x. For this reason, 32 is the default moderation factor in our I/O architecture. For IBufs tuning is more involved because we must prevent overflow of referenced DBufs. A simple and safe heuristic is to set the threshold to that of the smallest referenced DBuf.

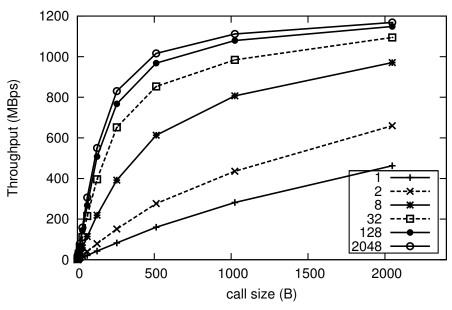


Figure 5.5: Interrupt moderation

5.3 Evaluation

In the previous sections we supported our claims by micro-benchmarks where applicable. We now compare our I/O architecture in a version of Linux 2.6.24.2 head-to-head with a stock version in terms of application throughput (or CPU utilization at steady rate). All tests were run on an HP 6710b with Intel Core 2 Duo T7300 processor, 4MB L2 cache and 2 GB RAM running in 32-bit mode. We ran the experiments on a single core to minimize scheduler influence and show the full cost of task switching.

5.3.1 Unix primitives

Figure 5.6 shows throughput of straightforward copying (a `write` followed by a `read`) through a Unix pipe at varying buffer size and for three IPC

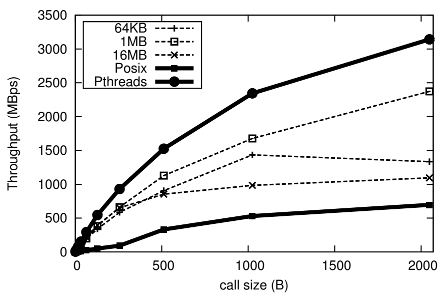


Figure 5.6: Unix pipe throughput: rate of several of our I/O architecture configurations compared to threads (best case) and processes (worst case).

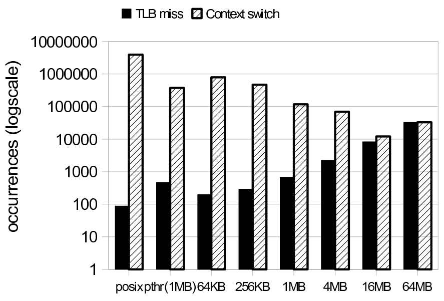


Figure 5.7: Pipe cost factors: number of task switches and cache misses observed at various buffer sizes.

implementations: a standard Linux pipe (with label 'posix'), a producer/-consumer pair of threads that directly access shared memory ('Pthreads') and buffers of varying size. In this test, we do not use the `peek` optimization and thus copy the same amount of data as the other applications. Any performance improvement comes from a reduction in context switching. The threaded application shows an upper bound on achievable performance, because it requires no kernel mode switches at all and it implements a multi-packet ring. Similar to our rings, its throughput is dependent on buffer size, but we only show the best case here for clarity (1MB). That configuration outperforms Linux's implementation of Unix pipes by a factor 5 for large blocks and 12 for minimal blocks. In between are 4 differently sized tail-drop DBufs. We see that the fastest implementation is neither the largest (64MB), nor the smallest (64KB), but an intermediate one (1MB). This outperforms Linux by a factor 4 for large and 9 for small packets and is only between 20 and 33% slower than the optimal case. The precise factor of throughput increase depends on physical cache size, producer-consumer distance and whether the application buffer is cached, but the ratios are static; we previously observed similar results on different hardware.

Figure 5.7 explains why the highest throughput is achieved with a medium-sized buffer. Initially, performance grows with the buffer as the number of necessary context switches drops when calls are less likely to block. Ultimately, however, page-faults affect performance as the datacache or TLB begins to witness capacity misses. These are more expensive than switches, therefore maximum throughput is obtained when the working-set just fits in the L2 cache.

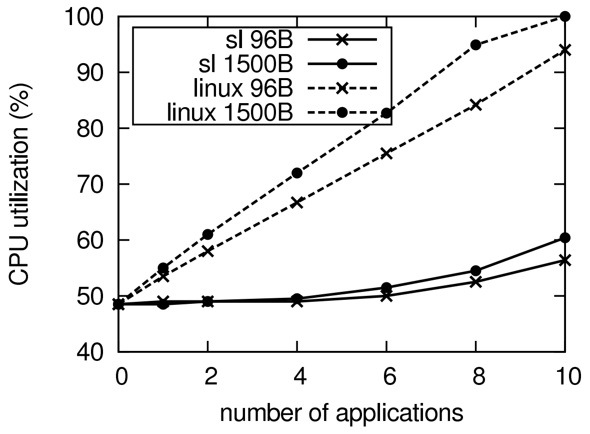


Figure 5.8: Tcpdump: CPU load for 200Mbps flow

5.3.2 Tcpdump and nemulator

The micro-benchmarks demonstrate that significant savings in I/O overhead can be achieved by optimizing buffer parameters and employing copy avoidance. We now investigate to what extent these savings translate to tcpdump and Nemu.

Figure 5.8 shows throughput of tcpdump 3.9.8, a popular traffic analyzer. To investigate scalability with parallel data access, we capture a moderate datastream: 200 Mbit of full-sized 1500B packets per second, generated with `iperf 2.0.2`. The iperf server requires 50% CPU time. When capturing with a single listener, ('sl 96B'), we uses up hardly any extra resources, while standard Linux ('linux 96B') requires 5% CPU time (10% of the application cost). Savings decrease as we run applications in parallel. When capturing full frames with 10 instances, we cause a 13% CPU overhead, slightly above a single Linux instance, whereas Linux saturates the CPU and drops packets. Running ten instances of tcpdump is not a common operation, but the results are representative for any system configuration where multiple applications access the same data, for instance network intrusion detection and group communication using multiway pipes.

Nemulator is a sophisticated detector of code injection attacks in the network based on Nemu. Rather than looking for specific patterns, Nemu detects an attacker's code (the shellcode) by treating every byte in the payload as its potential entry point. Thus, it will execute these bytes as if they were instructions. Typically, the execution hits an illegal instruction fairly quickly and Nemu will try again, with the next byte and so on. It raises an alert whenever the execution behaves in way that is indicative of shellcode (such as running GetPC sequences and executing bytes that it just wrote). Clearly, executing every byte in the payload is very expensive, and performance has been limited to a few tens of Mbps when checking the

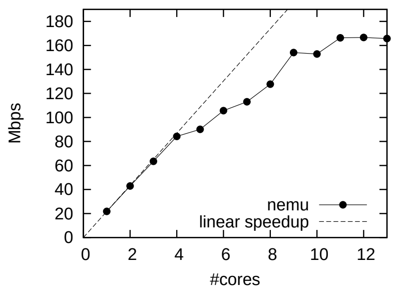


Figure 5.9: Nemu: multiple cores

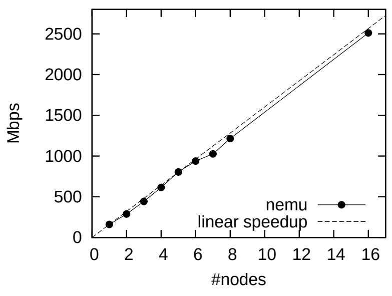


Figure 5.10: Nemu: multiple machines

full stream in this way. We built a parallelized Nemu implementation for multicore systems. A pipeline reassembles ingress TCP traffic and divides the connections among a fixed set of streams. Each stream is communicated by shared memory to a single core running a Nemu process. On a single Xeon X5650, 2.67GHz, 6-core machine with two threads per core, this configuration achieved 15-20 Mbps per thread and an overall throughput of 170Mbps with 11 processing threads. Figure 5.9 plots throughput with increasing numbers of cores. When spreading the task over multiple machines (replacing shared memory with UDP tunnels), the system scales to Gigabit rates. Figure 5.10 shows that this slightly different configuration observes linear scalability and reaches 2.5Gbps of aggregate throughput at 16 nodes.

5.4 Summary

By tailoring I/O paths automatically and on demand, we have taken an extreme position in the OS design space. The architecture reduces overhead from copying, context switching and caching, which improves throughput of a representative set of applications between 30% and 10x over standard Linux. Furthermore, it presents an OS-based solution to the problem of integrating special-purpose hardware. The result is practical software that can be, and has been, directly applied to networking tasks such as intrusion prevention, application serving and media streaming.

CHAPTER 6

Console

6.1 The Big Picture

As described in Chapter 1, i-Code project aims at detecting and analyzing malicious code and Internet attacks in real time. That's not, however, the only scope of this project: it also aims at creating an easy and centralized way to show the results of those tasks.

Therefore, while the underlying tools (described in the previous chapters) are the "core" of the system, great relevance must also be given to the only part that the user will interface with: the *console*. This component has to gather all the relevant events generated by the peripherical sensors and show them to the user in an understandable and usable way; the main result is that the tools become part of something bigger, that integrates and correlates information from different sources to provide the user with a clearer view of what's going on in the system. Thus, the component described in this chapter, though not being "active" within the system, has a fundamental role in i-Code project.

The i-Code console is a web application whose back-end performs the information retrieval from the database: this choice provides the user with a multiplatform environment accessible via browser by any kind of operating system or device. Thus, the front-end is very light and fast, while the whole logic is handled by the back-end, as described in Sections 6.2 and 6.3.

Figure 6.1 shows the architecture of the i-Code console and all its components.

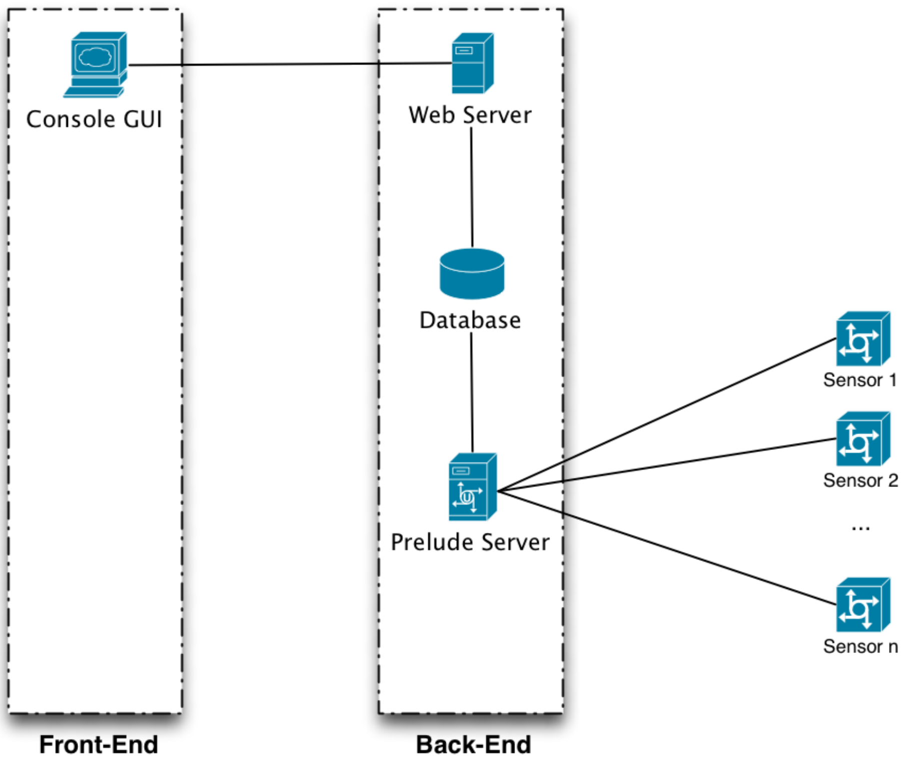


Figure 6.1: Overall architecture of i-Code console

6.2 Front-End

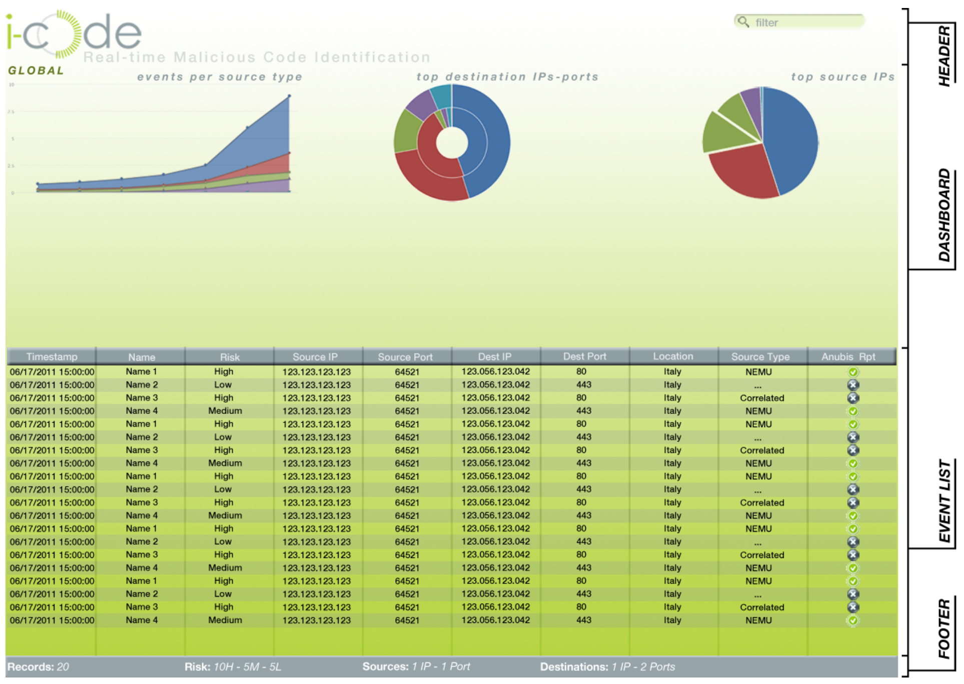
The front-end is what the user is shown as soon as he accesses the application; the initial view is simple, in order to provide an immediate feedback of the ongoing situation of the system: it shows charts on the general events, with no filtering, and lists them (see Figure 6.2).

The primary way the user can interact with the console is the *filtering system*: he can easily create filtering elements, called *filter tags*, and arrange them to compose complex queries, as described in Section 6.2.2.

6.2.1 Components

The front-end is composed of four components: header, dashboard, event list and footer; they are described in this Section.

Header The header allows to set and manage filters. Filter creation is described in Section 6.2.2 and modifies the main view as shown in Figure 6.3: the header shows the currently active filter set, new charts represent the



Timestamp	Name	Risk	Source IP	Source Port	Dest IP	Dest Port	Location	Source Type	Anubis Rpt
06/17/2011 15:00:00	Name 1	High	123.123.123.123	64521	123.056.123.042	80	Italy	NEMU	✓
06/17/2011 15:00:00	Name 2	Low	123.123.123.123	64521	123.056.123.042	443	Italy	...	✗
06/17/2011 15:00:00	Name 3	High	123.123.123.123	64521	123.056.123.042	80	Italy	Correlated	✗
06/17/2011 15:00:00	Name 4	Medium	123.123.123.123	64521	123.056.123.042	443	Italy	NEMU	✓
06/17/2011 15:00:00	Name 1	High	123.123.123.123	64521	123.056.123.042	80	Italy	NEMU	✓
06/17/2011 15:00:00	Name 2	Low	123.123.123.123	64521	123.056.123.042	443	Italy	...	✗
06/17/2011 15:00:00	Name 3	High	123.123.123.123	64521	123.056.123.042	80	Italy	Correlated	✗
06/17/2011 15:00:00	Name 4	Medium	123.123.123.123	64521	123.056.123.042	443	Italy	NEMU	✓
06/17/2011 15:00:00	Name 1	High	123.123.123.123	64521	123.056.123.042	80	Italy	NEMU	✓
06/17/2011 15:00:00	Name 2	Low	123.123.123.123	64521	123.056.123.042	443	Italy	...	✗
06/17/2011 15:00:00	Name 3	High	123.123.123.123	64521	123.056.123.042	80	Italy	Correlated	✗
06/17/2011 15:00:00	Name 4	Medium	123.123.123.123	64521	123.056.123.042	443	Italy	NEMU	✓
06/17/2011 15:00:00	Name 1	High	123.123.123.123	64521	123.056.123.042	80	Italy	NEMU	✓
06/17/2011 15:00:00	Name 2	Low	123.123.123.123	64521	123.056.123.042	443	Italy	...	✗
06/17/2011 15:00:00	Name 3	High	123.123.123.123	64521	123.056.123.042	80	Italy	Correlated	✗
06/17/2011 15:00:00	Name 4	Medium	123.123.123.123	64521	123.056.123.042	443	Italy	NEMU	✓
06/17/2011 15:00:00	Name 1	High	123.123.123.123	64521	123.056.123.042	80	Italy	NEMU	✓
06/17/2011 15:00:00	Name 2	Low	123.123.123.123	64521	123.056.123.042	443	Italy	...	✗
06/17/2011 15:00:00	Name 3	High	123.123.123.123	64521	123.056.123.042	80	Italy	Correlated	✗
06/17/2011 15:00:00	Name 4	Medium	123.123.123.123	64521	123.056.123.042	443	Italy	NEMU	✓

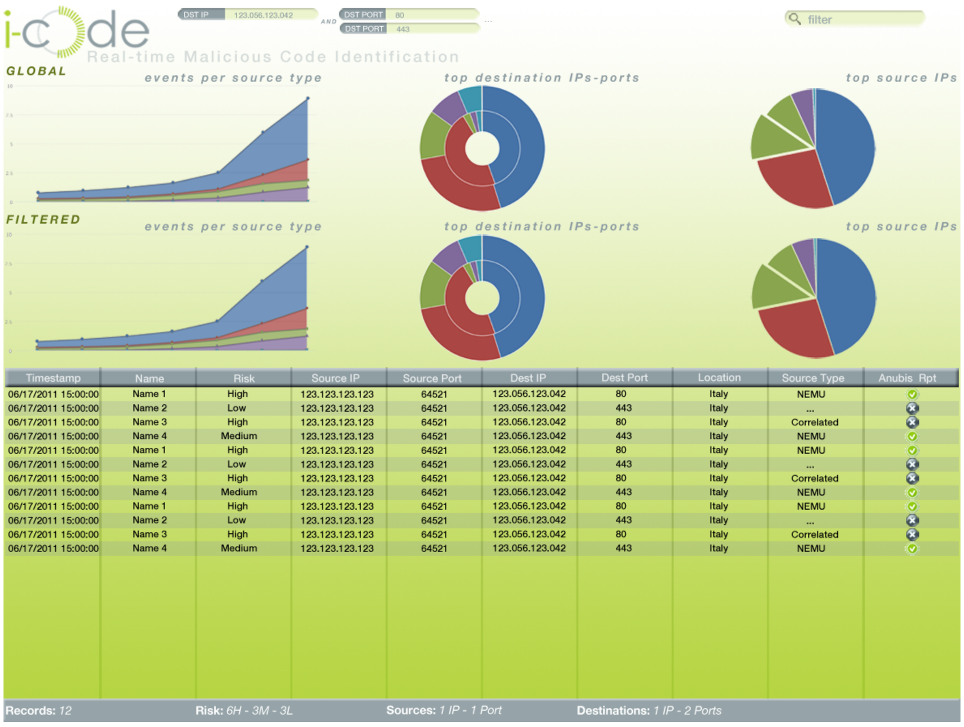
Records: *20* **Risk:** *10H - 5M - 5L* **Sources:** *1 IP - 1 Port* **Destinations:** *1 IP - 2 Ports*

Figure 6.2: Mockup of the i-Code console's main screen, composed of four parts: header, dashboard, event list and footer.

situation of the system with respect to the only events the user is interested in, and the event list is restricted to those events only.

Dashboard The dashboard shows a graphical overview of the alerts. In particular, three charts are shown: daily number of events per source type, top destination IPs and ports, and the top source IPs. The console displays this information both for the entire dataset (upper section of the dashboard) and for filtered events (lower section). If no filter is selected, only the upper section will be visible, as shown in Figure 6.2.

Event list This part shows the events that match the desired filter, if any; otherwise it reports all the events in the database. Every row shows date and time of the event, its name (if known), risk (high, medium or low), source IP and port, destination IP and port, and country of origin. The source type is also reported, so that the user will know what tool (e.g., *Nemu*) sent the event. In addition, the last column shows whether an Anubis report is available for the event. If available, the report is shown whenever the user clicks on the green icon. Otherwise the click triggers a request sent to Anubis, which generates the report. This behavior provides the system with the event forwarding capability that was part of the initial requirements defined in deliverable *D1: System Design*.



Timestamp	Name	Risk	Source IP	Source Port	Dest IP	Dest Port	Location	Source Type	Anubis Rpt
06/17/2011 15:00:00	Name 1	High	123.123.123.123	64521	123.056.123.042	80	Italy	NEMU	✓
06/17/2011 15:00:00	Name 2	Low	123.123.123.123	64521	123.056.123.042	443	Italy	...	✗
06/17/2011 15:00:00	Name 3	High	123.123.123.123	64521	123.056.123.042	80	Italy	Correlated	✗
06/17/2011 15:00:00	Name 4	Medium	123.123.123.123	64521	123.056.123.042	443	Italy	NEMU	✓
06/17/2011 15:00:00	Name 1	High	123.123.123.123	64521	123.056.123.042	80	Italy	NEMU	✓
06/17/2011 15:00:00	Name 2	Low	123.123.123.123	64521	123.056.123.042	443	Italy	...	✗
06/17/2011 15:00:00	Name 3	High	123.123.123.123	64521	123.056.123.042	80	Italy	Correlated	✗
06/17/2011 15:00:00	Name 4	Medium	123.123.123.123	64521	123.056.123.042	443	Italy	NEMU	✓
06/17/2011 15:00:00	Name 1	High	123.123.123.123	64521	123.056.123.042	80	Italy	NEMU	✓
06/17/2011 15:00:00	Name 2	Low	123.123.123.123	64521	123.056.123.042	443	Italy	...	✗
06/17/2011 15:00:00	Name 3	High	123.123.123.123	64521	123.056.123.042	80	Italy	Correlated	✗
06/17/2011 15:00:00	Name 4	Medium	123.123.123.123	64521	123.056.123.042	443	Italy	NEMU	✓

Records: *12* **Risk:** *6H - 3M - 3L* **Sources:** *1 IP - 1 Port* **Destinations:** *1 IP - 2 Ports*

Figure 6.3: Mockup of the i-Code console showing the events matching a custom filter (in this dummy example, all the events having destination IP equal to 123.056.123.042 and destination port equal to 80 or 443). Note that a filter is created by arranging filter tags in the header section and, once set, the dashboard shows three additional graphs reflecting this smaller portion of data.

Footer The footer summarizes the events in the event list, providing the user the total number of records matching the filter, their risk and the number of distinct source and destination IPs and ports.

6.2.2 Filtering system

The main functionality implemented by the front-end is the filtering system: it allows to define complex filters to narrow event searches down and to display only events satisfying some criteria, thus greatly improving the usability of the i-Code console.

Its design is quite simple: using the input field in the top right corner of the console, the user creates the atomic units of the filters, called *filter tags*; they are simple key-value pairs (e.g., src-ip = . . .) expressing a condition and they can be created over any of the fields that describe an event. Also, auto-completion is provided to help the user create filter tags with the least effort.

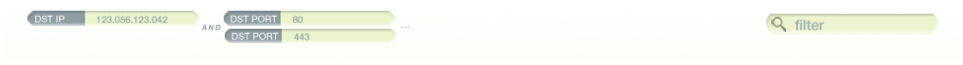


Figure 6.4: Close-up of the filtering system

Once a user defines a *filter tag*, it is placed in the console header and that's where it gets effective in the system: every tag is evaluated based on its position with respect to other tags. In this way complex filters can be created by means of simply dragging tags in the correct place.

In particular, filter tags in rows will be conjuncted (i.e., condition_set1 *and* condition_set2) while those in columns will be disjuncted (i.e., condition_set1 *or* condition_set2). Figure 6.4 shows an example of a filter, which can be read as "show all the events having destination IP equal to 123.056.123.042 *and* destination port equal to 80 *or* 443."

Technology As for the rest of the i-Code console, the filtering system has to be usable on every devices that has web capabilities (i.e., that can run a fully–functional web browser): that's why, wherever possible, no proprietary technologies or standards were used.

AJAX was therefore the main choice to build the filtering systems and, at the same time, allowing a smooth user experience while using the console: AJAX, in fact, allows to compose and update filters, retrieve data from the back-end and show events in an asynchronous way, without having to completely reload the page (a quite annoying drawback of standard HTML environments).

6.3 Back-End

The i-Code console back-end is where all the data and logic reside, but not only: it also includes the overall infrastructure built to gather, parse and store new events.

The back-end is therefore made of two distinct parts: the web application and the event collector.

6.3.1 Web Server and Database

The web application back-end implements all the methods tailored at providing the user with all the required data. In the case of the i-Code console it is quite simple, since the only relevant task is performing the desired query on the database and returning the results to the user.

The only bottleneck in this approach is the database itself (if it contains many events) but there are plenty of ways to solve the issue, creating the correct indexes being the simplest one.

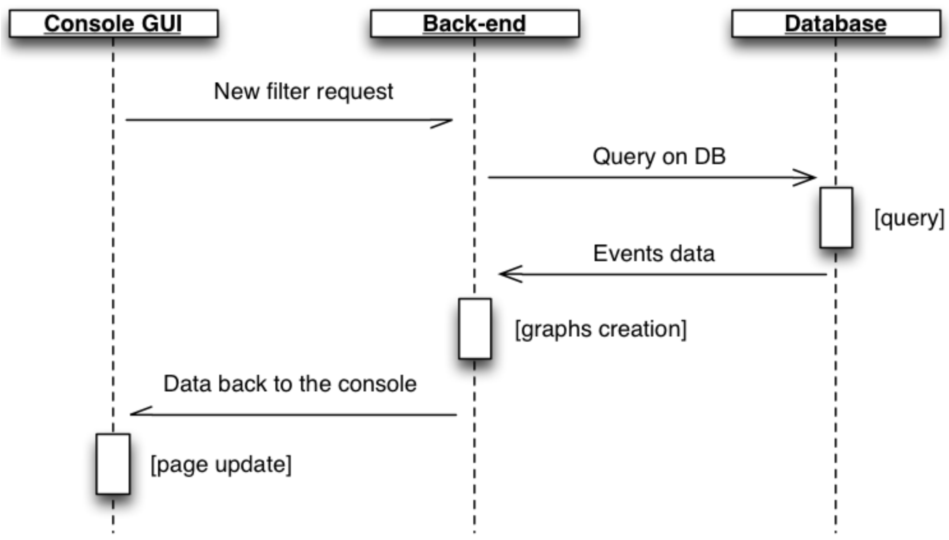


Figure 6.5: Sequence diagram of the filter update

Figure 6.5 shows the sequence of actions required to update a view after the user has updated a filter: note that the web server has also to create the new graphs to be displayed on the main page of the GUI.

This part of the back-end is built using Django and MySQL. The first is a framework tailored at creating web applications and therefore perfectly suitable for our purpose; it is also highly customizable and lightweight. The database, instead, is a standard MySQL server, compatible with both the web application and the Prelude MySQL connector.

6.3.2 Prelude

The i-Code console includes a component that has the role of aggregating and storing all the events coming from the peripherical sources (see Figure 6.1); instead of building it from scratch, we decided to rely on an existing working application: Prelude[1].

Prelude is built around a centralized manager which handles all the incoming events and sends them to the database and, possibly, to the correlation engine(s). The manager, which can be made redundant for the sake of availability, is the core of the whole system, since all the communications, event handling, notifications and event storing are on it. It is, anyway, possible to deploy its components on different machines, to lower computational load on a single machine.

[1]http://www.prelude-technologies.com/

The external components are responsible for the generation and the dispatching of security events and, excluding particular cases (e.g. the correlation engine), have to be deployed on the monitored machine. As mentioned, there are “special” components that can be instead deployed anywhere, since they only need access to the database and the manager; it’s the case of the correlation engine, which does not generate events from a host machine but rather analyzes all the events gathered by the system and creates new ones based on custom rules.

Communication between the manager and all the components (both central and peripheral) is secured via asymmetric cryptographic keys which are automatically exchanged during the component registration phase.

Infrastructure

Prelude includes four main components:

Manager It’s the core of the system, as already described. It requires libPrelude to work correctly, since it uses this library to handle secure connections with the components.

Event storage connector The manager does not store the events on its own: to make it fully customizable, it was built to use an external connector component that manages the event storage. There is a different component for every type of storage, be it a simple file on the hard drive or a database or even a SNMP notification; its task is to reserve and maintain all the necessary resources to reliably store the events and make them available when (and if) needed. In the case of the i-Code console, a MySQL connector is used, in order to be able to read and write on the database included in our architecture.

Correlator The correlator engine is a particular kind of sensor: it generates events like all other sensors, but they do not originate from some security alerts. They are rather the result of an internal rule matching: this components gets the events from the manager and checks them against some precompiled rules; if a rule triggers a new event is raised and sent to the manager.

External sensors These are the components that are responsible of generating events based on internal mechanisms; sensors can be of any kind, depending on the requirements of the system: they can be authentication sensors, intrusion detection systems, physical access sensors, . . . In i-Code the only peripherical components that will be used are the ones described in the previous Chapters.

Database Format

Prelude system exchanges events among the components in *IDMEF* (Intrusion Detection Message Exchange Format) format. IDMEF is based on XML and can be completely customized to someone's needs. In Prelude, there are five types of events: *Alert*, *CorrelationAlert*, *OverflowAlert*, *ToolAlert* and *Heartbeat*. All the events have the structure defined in Table 6.1 (names in square brackets refer to nodes, not simple fields).

i-Code and Prelude

Prelude has some limitations due to the event format it uses by default and the necessity to build compatible sensors in order to deploy them within the system.

In the previous section we described the event format used by Prelude. For i-Code, instead, the event format is designed to include the following fields:

- Timestamp
- Name
- Risk
- Source IP
- Source Port
- Destination IP
- Destination Port
- Location
- Source Type

The two obviously slightly differ. In order to use the i-Code format, the most correct way would be to learn how Prelude handles the event fields in its source code and how they are then mapped onto the database; this would require, though, a deep study of the source code of the manager component and might lead to modifications that cannot be done in an easy and reliable way. If modifications are applied on the source code without being well planned, they might lead to bugs or break the compatibility with existent component.

The adopted workaround is to use the *Additional Data* field in Prelude format to add all the data that i-Code has that are not easily mappable in the available IDMEF format. This field accepts data in the format $< label, value >$, so multiple occurrences can exist within a single event. The

Node	Fields
AdditionalData	type meaning data
Address	category vlan_name vlan_num address netmask
Analyzer	analyzerid name manufacturer model version class ostype osversion
AnalyzerTime	time usec gmtoff
Assessment	*[Confidence]* [Impact]
Checksum	algorithm value checksum_key
Classification	text
Confidence	confidence rating
CreateTime	time usec gmtoff
DetectTime	time usec gmtoff
File	path name category create_time create_time_gmtoff modify_time modify_time_gmtoff access_time access_time_gmtoff data_size disk_size fstype file_type
FileAccess	*[File]*

Node	Fields
Impact	description severity completion type
Node	*[Address]* category location name
Process	*[ProcessArg]* *[ProcessEnv]* name pid path
ProcessArg	arg
ProcessEnv	env
Reference	origin name url meaning
Service	ip_version name port iana_protocol_numer iana_protocol_name portlist protocol
Source	*[User]* *[Service]* *[Node]* *[WebService]* spoofed interface
Target	*[User]* *[Service]* *[Node]* *[WebService]* decoy interface
User	*[UserId]* category
UserId	type name tty number
WebService	*[WebServiceArg]* url cgi http_method
WebServiceArg	arg

Table 6.1: Prelude IDMEF format

correct mapping is the one proposed in Table 6.2. Note that this is the minimum requirement in an event: other fields can be added when necessary.

i-Code field	**Prelude field**
Timestamp	alert.detect_time.time
Name	alert.classification.text
Risk	alert.assessment.impact.severity
Source IP	alert.source(0).node.address(0).address
Source Port	alert.source(0).service.port
Destination IP	alert.target(0).node.address(0).address
Destination Port	alert.target(0).service.port
Location	alert.additional_data(0).type = 'string' alert.additional_data(0).meaning = 'location' alert.additional_data(0).data = [value]
Source Type	alert.analyzer.model

Table 6.2: Mapping from i-Code event fields to Prelude IDMEF format

CHAPTER 7

Testing Phase

After the development phase of the i-Code project, the single components (both sensors and console) will be integrated and the overall system will undergo a testing phase.

The aim of this phase is to verify if the single components work correctly and, more important, if their integration is correct and flows as expected. Figure 7.1 shows the basic integration of the i-Code components in a complete system.

As depicted in Chapter 6, the console is the collector of all the events generated by the sensor: in particular it receives the security events from *Argos* [18], *NEMU* (described in Chapter 2) and *Access Miner* (described in Chapter 4). These events are directly imported in the database and shown to the user but they're not the only data exchanged between the sensors and the console; in the case of *Argos* and *NEMU*, in fact, the events are coupled with the shellcode that originated the alert: this chunk of data is then sent by the console to the *Anubis* remote system, where it is analyzed. When done, *Anubis* sends a report id to the console, which is then saved in the appropriate field of the security event and shown in the GUI. The same interaction happens between *Access Miner* and *Anubis*; in this case, though, the sensor itself sends the sample it collected to *Anubis* and waits for a report id: when it possesses all these information, it sends the alert and the report id directly to the console.

The console, however, is not a simple receptor of the event: during the testing phase some basic correlation rules will be defined and implemented. They will be the outcome of a manual analysis of the behavior of the system in order to reduce the ”noise” of the events; one of these basic rules will be

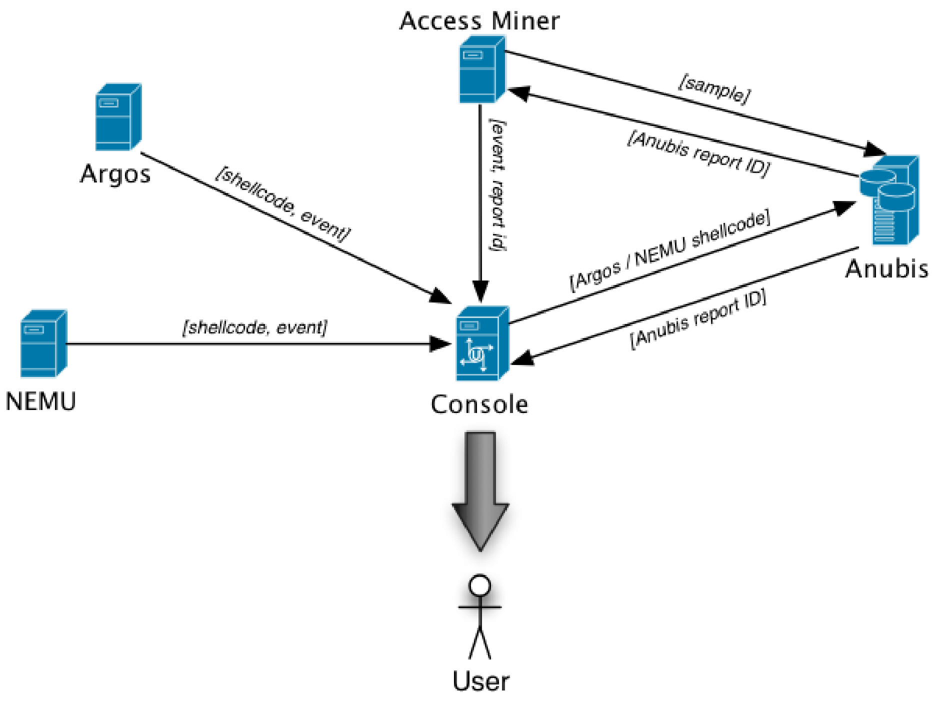


Figure 7.1: Interactions among components

therefore the aggregation (and removal) of duplicate events among different sensors, a situation that is expected to be frequent with *Argos* and *NEMU*.

The testing phase will be delivered in two distinct steps, in different environments and with different purposes: the first one will be done in a completely virtualized environment and its aim will be to test the main functionalities of the system, while the second testbed will be more similar to a real development.

7.1 First testbed

The first testbed will work as a testbench for the basic functionalities of the i-Code system, to see if everything works correctly and the integration of the single components flows as expected.

The architecture of the virtualized environment is shown Figure 7.2: six virtual machines will be installed, each of them in bridge mode and with different IP addresses. *Access Miner* will be installed on two of them, while one will host *Argos*. *NEMU* will be installed on a yet different machine and will be set in promiscuous mode to let it sniff the entire traffic flowing through the virtual network. The remaining two virtual machines will be dedicated

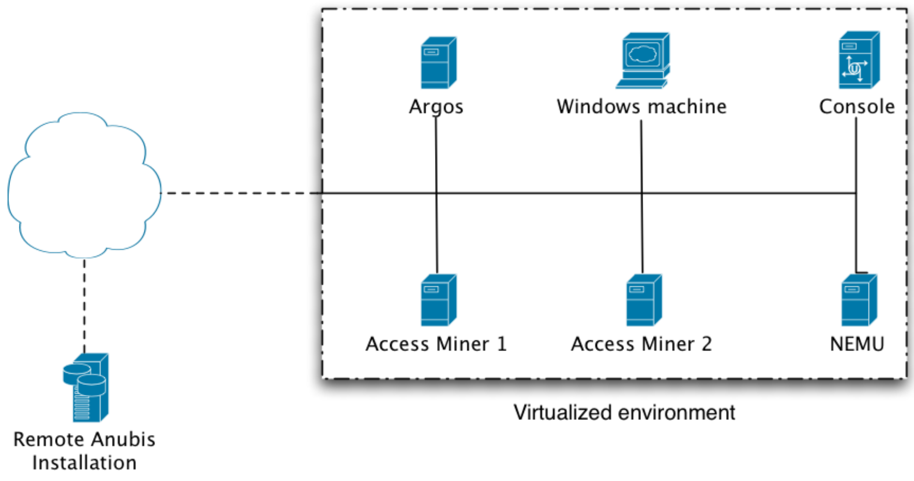


Figure 7.2: Architecture of first testing environment

to the console (including the entire backend described in Section 6.3) and to a Windows machine used to generate security events.

7.2 Second testbed

The second step of the testing phase will be conducted in a real environment, with both the sensors and the console fully functional.

This time, though, *NEMU* will run on a machine located at FORTH, while *Argos* will be hosted on a virtual machine in VU.

CHAPTER 8

Conclusions

In this deliverable, we described the implementation of the i-Code real-time malicious code detection system. In particular, we discussed the implementation of our tools and sensors that are able to detect and analyze malicious code and Internet attacks in real time. These sensors perform shellcode detection based on network level emulation, the behavioral analysis of shellcode and the subsequent classification based on the structure of its unpacked code, and the detection of malware on the end host by contrasting its behavior against normal behavior patterns from real, uncompromised machines. Furthermore, we discussed the implementation of a scalable, high/performance I/O architecture that we developed to speed up payload execution.

We also discussed the implementation of the i-Code console that gathers all the relevant events generated by the sensors and presents them to the user in an understandable and usable way. Therefore our tools and sensors become part of something bigger, that integrates and correlates information from different sources to provide the user with a clearer view of what's going on in the network.

In the remainder of the i-Code project we will work on the integration of the implemented components (both sensors and console) into a comprehensive and unified system. We will then evaluate this system in two testing phases in order to verify its functionality and effectiveness. The first, preliminary testing phase will be performed in a virtualized environment. The second testing phase will evaluate our system in a real environment.

Bibliography

[1] cluster: Cluster Analysis Extended Rousseeuw et al. http://cran.r-project.org/web/packages/cluster.

[2] fastcluster: Fast hierarchical clustering routines for R and Python. http://cran.r-project.org/web/packages/fastcluster.

[3] The R Project for Statistical Computing. http://www.r-project.org/.

[4] Anubis. 2007. http://anubis.seclab.tuwien.ac.at.

[5] P. Bania. TAPiON, 2005. http://pb.specialised.info/all/tapion/.

[6] U. Bayer, P. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, 2009.

[7] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A Tool for Analyzing Malware. In *Proceedings of the 15th European Institute for Computer Antivirus Research (EICAR) Annual Conference*, 2006.

[8] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference*, 2005.

[9] jt. Libdasm, 2006. http://www.klake.org/~jt/misc/libdasm-1.4.tar.gz.

[10] C. Kolbitsch, P. Milani, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th USENIX Security Symposium (Security'09)*, pages 351–366, Montréal, Canada, Aug. 2009. USENIX Association.

[11] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Symp. on Recent Advances in Intrusion Detection (RAID)*, 2005.

[12] J. Ma, J. Dunagan, H. J. Wang, S. Savage, and G. M. Voelker. Finding diversity in remote code injection exploits. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, pages 53–64, 2006.

[13] S. McCanne, C. Leres, and V. Jacobson. Libpcap, 2006. http://www.tcpdump.org/.

[14] L. McVoy. The splice I/O model. www.bitmover.com/lm/papers/splice.ps, 1998.

[15] D. Mueller. fastcluster: Fast hierarchical clustering routines for R and Python, 2011. http://math.stanford.edu/~muellner/fastcluster.html.

[16] M. Polychronakis, E. P. Markatos, and K. G. Anagnostakis. Network-level polymorphic shellcode detection using emulation. In *Proceedings of the Third Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, July 2006.

[17] M. Polychronakis, E. P. Markatos, and K. G. Anagnostakis. Emulation-based detection of non-self-contained polymorphic shellcode. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2007.

[18] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. In *Proc. ACM SIGOPS EUROSYS'2006*, Leuven, Belgium, April 2006.

[19] R. Wojtczuk. Libnids, 2006. http://libnids.sourceforge.net/.