2012 Seventh International Conference on P2P, Parallel, Grid, Cloud and Internet Computing

# Parallel Perspectives: Reverse Engineering for Generation Multi-X

Andreas Bergen*, Dean Pucsek*, Jennifer Baldwin*, Laura MacLeod*
Celina Berg*, Martin Salois†, Yvonne Coady*
*University of Victoria
Email: {andib, dpucsek, jbaldwin, lmacleod, celinag, ycoady}@cs.uvic.ca,
†Defence Research and Development Canada–Valcartier
Email: martin.salois@drdc-rddc.gc.ca

***Abstract*—Reverse engineering large systems today requires multiple analysts trying to understand multi-threaded software running on multicore/multiprocessor hardware that covers multiple instruction sets. Today's "multi-X" world requires new analysis tools revealing a wide range of perspectives—static and dynamic, detailed and abstract. Our goal is to design tools to support low-level program comprehension tasks ranging from malware analysis to mainframe code-base evolution.**

**This paper presents the design and implementation of a prototype Integrated Comprehension Environment (ICE) with multiple analysis plugins designed to provide reverse engineers with the perspectives they need to analyze today's software. Leveraging the simplicity of the Reverse Engineering Intermediate Language (REIL), this prototype provides plugin modules designed to correlate low-level and high-level perspectives of both static and dynamic information. Our goal is to determine if this generalized design may in fact enable tool sets to be unhinged from specific instruction sets, permitting the same analysis modules to be applied to code bases made for drastically different architectures, such as x86 versus HLASM.**

***Keywords*-software reverse engineering, parallel, SIMD**

## I. Introduction

Understanding and analyzing program behaviour is challenging, even when inspecting source code written in high-level languages. When required to perform this analysis on code bases solely represented in assembly language, comprehension becomes even more challenging. For example, inspecting binaries that leverage parallelization, or worse still, self-modifying code—commonly to both malware and mainframe programming— is very difficult.

In terms of tools designed to analyze low-level code, IDA Pro [1] is the industry standard disassembler, offering up binaries as low-level assembly programs for manual analysis and integration with automated tool support. CodeSurfer [2] is one example that leverages some of the strengths of IDA Pro. Another tool for binary analysis is Bitblaze [3], which is a research platform that provides a unified and extensible analysis infrastructure. Comprehensive techniques used in binary analysis of x86 executables are explored and illustrated by Kinder [4].

With the onslaught of new architectures and extended instruction sets, increasing the level of abstraction to a common intermediate representation has become a popular approach. Intermediate representations such as REIL [5] and LLVM [6] provide a slightly higher level representation than that of assembly. The simplicity of REIL's instruction set and the extensibility of LLVM in terms of adding new analysis tools are making them a viable choice for program analysis. Both REIL and LLVM support translation from a high-level program representation to their intermediate representation but the challenge of systematically translating from low-level assembly into to these higher level representations is subject to tradeoffs, as information that may be needed by analysis tools could be lost in this translation.

The work presented here focuses on our proposed Assembly Visualization and Analysis (AVA) Framework, exploring the applicability of high-level visualization and analysis tools to low-level code bases [7]. This research project includes requirements elicitation to develop tools that are needed both in a security context for malware analysis, as well as for comprehension, maintenance and evolution of complex applications in legacy mainframe systems. So far the framework consists of tools that support control flow [8] as well as documentation within that control flow [9]. Since the goal of this project is to support assembly language in general, it is important that it remains agnostic in terms of instruction sets. Unfortunately, the reality is that it is already difficult for the creators of these tools to extract the detailed data they need, especially when taking into account the multiple assembly language dialects in existence. While AVA's first aims are to support x86 and HLASM [10], this support would ideally be based on a common intermediate language. This intermediate language could then be leveraged by developers wishing to have this support for their own assembly language dialect. All they would need is to create their own translator to bridge into the intermediate representation.

With the Integrated Comprehension Environment (ICE) [11], we proposed the concept of an extensible framework for the purpose of program navigation and analysis. This work offered a case study at the level of preprocessor directives in C-based systems. The work presented here continues along these lines but focuses on lower-level code bases. In this paper, Section II provides an overview of ICE and the ways in which it incorporates a growing set of analysis tools targeting both the static and dynamic comprehension of low-level programs. Section III considers the extensions required for ICE to be used within parallel environments and Section IV provides a discussion and conclusion.

## II. Integrated Comprehension Environment (ICE)

To assist a reverse engineer in the process of binary analysis, we have developed a prototype tool called *Integrated Comprehension Environment (ICE)*. ICE has been designed from the ground up to support multiple simultaneous analyses as well as visualizations of the binaries being analyzed. ICE allows a reverse engineer to define a customized analysis tool by writing a module in Python that can leverage an internal API that provides access to various entities commonly found in programs such as instructions, segments, and functions. In addition, ICE does not limit the module developer to a single thread. ICE is able to spawn threads and analyze the binaries from all of them.

When a binary is opened inside ICE, the machine code is translated into an intermediate language. The primary advantage of this approach is that analysis modules need only be written for a single language, the intermediate language, while enabling analyses to be carried out on a wide range of instruction sets. In addition, this lowers the overhead of supporting new architectures as they become relevant (only a translator needs to be written).

While there is a plethora of intermediate languages already available, we chose Reverse Engineers Intermediate Language (REIL) [5]. The decision to use REIL was based on two things: simplicity and extensibility. REIL is a small, 17 instructions, RISC pseudo-assembly language developed by a company called Zynamics. Six of the instructions in REIL are arithmetic which include standard operations: `add`, `sub`, `mul`, `div`, `mod`, and `bsh` (bitwise shift). Multiply and divide are unsigned operations, the remaining arithmetic instructions are signed.

With respect to bitwise operations, REIL contains `and`, `or`, and `xor`. Note that bitwise-`not` is not included because it can be represented using a bitwise-`xor`. Data transfer may only occur via three instructions: `str` (register transfer), `ldm` (load from memory), and `stm` (store to memory). The only branch instruction in REIL is `jcc` (conditional jump) which is typically used in conjunction with the `bisz` instruction that checks the source operand for a zero value and places a `0` or `1` in the destination operand accordingly. In addition, REIL provides an `unkn` instruction to indicate that a translation is not possible (for example, the translator simply did not provide a definition for the corresponding machine instruction) and an `undef` instruction that allows a register to be placed in an undefined state. Finally, REIL provides a `nop` instruction to represent no operation. In addition to the instructions, REIL also provides an unlimited number of registers and does not limit the range of addressable memory. The extensibility of REIL allows for new instructions to be added as necessary to support complex CPU extensions such as hardware virtualization.

```
; eax = *(esp+4) [C syntax]
0x001  mov      eax, [esp+4]
0x002  next:    cmp     byte ptr [eax], 0
0x003  je               done
0x004  inc      eax
0x005  jmp      next
;; eax = eax - [esp+4]
0x006  done:    sub     eax, [esp+4]
0x007  ret
```

Figure 1. An x86 implementation of a simple string length function

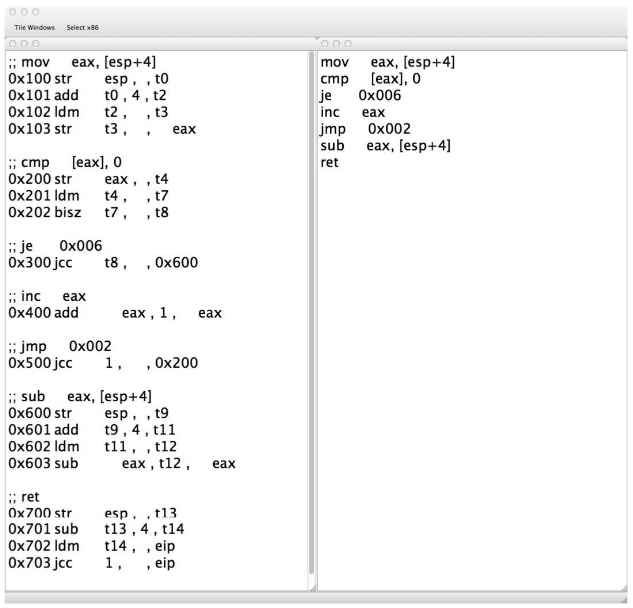
The following two subsections describe the static and dynamic tools that interface with the ICE framework, providing a variety of both detailed and abstract perspectives.

### A. Static Analysis

*1) Detailed Landscapes:* To demonstrate the static analysis capabilities integrated with ICE, we consider the simple x86 code in Figure 1, implementing a string length computation algorithm. The code compares each character to NULL (`0x002`) and increases the memory pointer if they are not equal (`0x004`) and repeats (`0x005`). Upon finding `NULL` (`0x003`), the code computes the difference between the address of the original string pointer and the modified string pointer to get the string length (`0x006`) then returns (`0x007`). Note that this simple, seven-instruction implementation does not contain the additional instructions required to execute it.

Figure 2 demonstrates the detailed static analysis perspective provided by the integration of a static x86 to REIL translator within ICE. This figure shows the string length function of Figure 1 in both the original assembly (right hand side) and its translation to REIL (left hand side). Note that the hexadecimal values for the `jmp` and `je` instructions are the corresponding jump targets—the control will go to this address should the `jmp` condition be true.

In the REIL translation, we have left the original x86 instruction as a comment at the beginning of each group of REIL instructions to aid the reader in inferring the translation. Furthermore, to help with programmatically correlating REIL instructions back to the original x86 instruction, the original address is shifted left by one byte and an offset into the REIL translation of the current instruction is placed in the newly available byte thus enabling the original instruction to be found by masking the REIL address with `0xFF`. In addition, the memory references in x86 become explicit



```
;; mov     eax, [esp+4]
0x100 str       esp ,   , t0
0x101 add       t0 , 4 , t2
0x102 ldm       t2 ,    , t3
0x103 str       t3 ,    ,     eax

;; cmp     [eax], 0
0x200 str       eax ,   , t4
0x201 ldm       t4 ,    , t7
0x202 bisz      t7 ,    , t8

;; je     0x006
0x300 jcc       t8 ,    , 0x600

;; inc     eax
0x400 add           eax , 1 ,    eax

;; jmp     0x002
0x500 jcc       1 ,     , 0x200

;; sub     eax, [esp+4]
0x600 str       esp ,   , t9
0x601 add       t9 , 4 , t11
0x602 ldm       t11 ,   , t12
0x603 sub           eax , t12 ,    eax

;; ret
0x700 str       esp ,   , t13
0x701 sub       t13 , 4 , t14
0x702 ldm       t14 ,   , eip
0x703 jcc       1 ,     , eip
```

```
mov     eax, [esp+4]
cmp     [eax], 0
je      0x006
inc     eax
jmp     0x002
sub     eax, [esp+4]
ret
```

Figure 2. ICE static results.

in REIL. That is, memory may only be accessed through store (`stm`) and load (`ldm`) instructions, allowing the analyst to more easily track memory usage. Finally, as a result of the more explicit nature of REIL, it is quite common for a single x86 instruction to be translated into numerous REIL instructions.

*2) Abstract Vistas:* While it is relatively easy to understand a few lines of assembly, the problem is much more difficult when trying to understand the millions of lines of code in typical programs. One quickly gets lost, especially if the code is obfuscated in any way. We believe detailed views of these types of code bases must be augmented with abstract views and ultimately the integration of these views would occur at the level of the ICE framework.

To deal with this cognitive overload with source code, Robert DeLine [12] attempted to tap into the human's innate ability to understand a map. Out of this work, we developed a prototype for assembly visualization named MapUI, a visualization tool that uses a spatial memory approach in hopes of reducing redundant navigation and increasing code comprehension. MapUI creates a software terrain by breaking code into "countries" whose resulting shapes create visual landmarks. This is demonstrated in Figure 3. A "country" is a function with the size defined by the number of lines of assembly. Proximity is determined by a measure of affinity defined by the number of calls between the functions.

Figure 4 shows an example of MapUI displaying a *crackme* [13], a small program designed to practice reverse engineering. Crackmes are designed to be difficult to analyze in key areas. The resulting terrain map shows a program

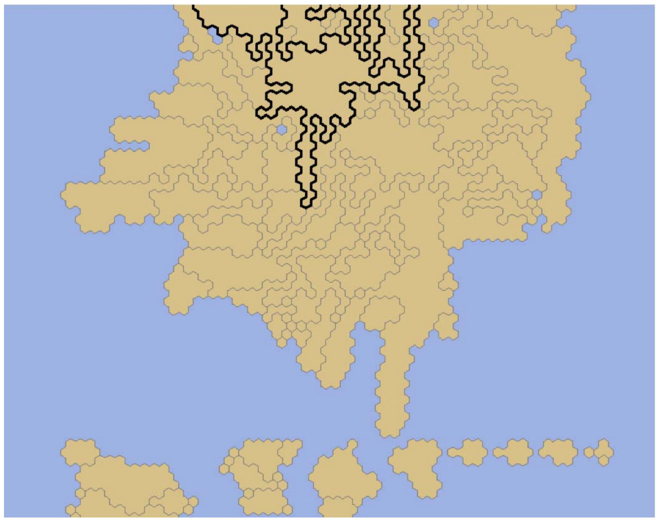


Figure 3. MapUI, running in conjunction with IDA Pro, displaying a crackme program.

with many irregularly shaped components. Clicking on a component highlights it and, if IDA Pro is running, it jumps to the corresponding location in the assembly code. In MapUI, continents or islands are created when there are no static links to other functions (e.g. exception handling, dead code). Colours can be applied and layers can be created according to different concepts.

The user does not have to rely on detailed flow charts to identify the general connections between portions of code and can create groupings that work for their purpose. Using MapUI, one can quickly determine how pieces of code interact and determine adjacent components. But the main point is that it is easier for a human to remember what country he was in a week later than at what hexadecimal address. Beyond that, further studies are required to determine the actual usefulness of the prototype.

## B. Dynamic Analysis

We continue with the string length code sample to demonstrate the dynamic analysis tools that plug into the ICE framework, providing again both detailed and abstracted views of the dynamic output.

*1) The devil is in the details:* Here we showcase the simulator, designed and implemented to simulate the execution of a program represented in REIL.

Consider again the simple implementation of the string length algorithm shown in Figure 1. The left-hand side of Figure 2 is the input to the simulator interfacing with ICE in Figure 5.

After parsing REIL code, the simulator determines the program entry point, simulates the execution of the current instruction and then either terminates or locates the next instruction. Figure 6 shows the raw output of the simulator,

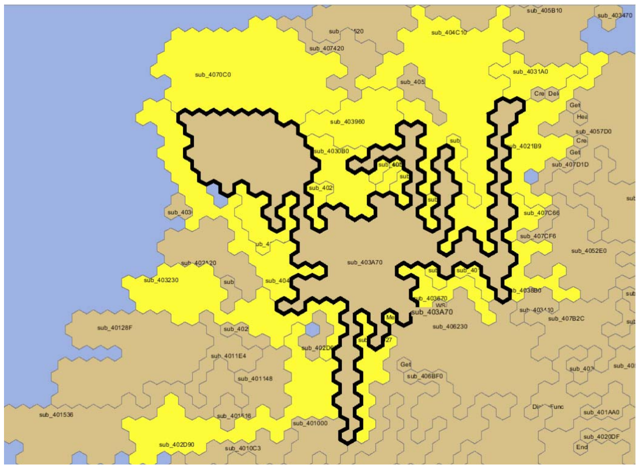


Figure 4. Coloring applied to adjacent components of sub_403A70.

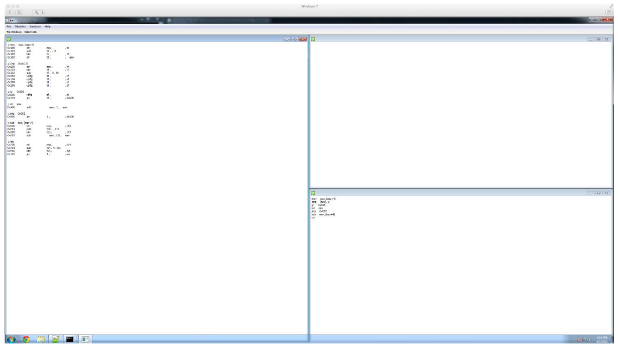
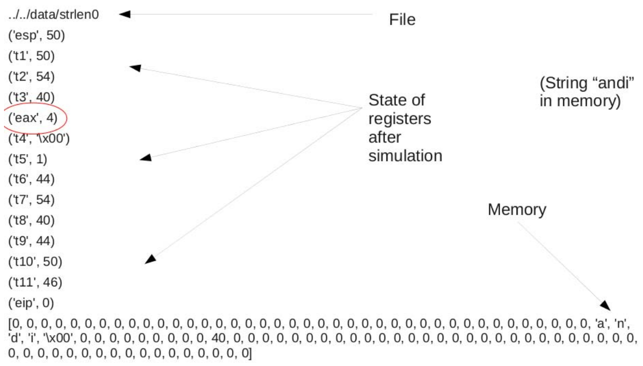


Figure 5. ICE and REIL simulator: dynamic analysis and results


File
State of
registers
after
simulation
(String “andi”
in memory)
Memory

```
../../data/strlen0
('esp', 50)
('t1', 50)
('t2', 54)
('t3', 40)
('eax', 4)
('t4', '\x00')
('t5', 1)
('t6', 44)
('t7', 54)
('t8', 40)
('t9', 44)
('t10', 50)
('t11', 46)
('eip', 0)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 'a', 'n',
'd', 'i', '\x00', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 40, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Figure 6. Raw text output from the simulator (return register circled red for illustrative purposes)

which includes register values and the memory state after the program simulation is complete. The return value is circled in red here to highlight the return register's value.

The simulator can capture every side effects on the system. That is, any change that is inflicted upon any part of the simulated environment is recorded. Data flow, taint information, dynamic control flow and call graphs are all elements that can be recorded and displayed in text format.

As such, the simulator lends itself also to exhaustive comparisons of various runs of REIL code. Similar to fuzz testing, one can compare the effects on the simulated system given deviating input sets to the REIL program through the simulator.

*2) Answering with another layer of abstraction:* Interfacing the simulation's output with AVA's Tracks visualization tool [8], a tool developed to assist with the comprehension of large assembly codebases, we introduce another layer of abstraction to assist with the comprehension of dynamic analysis results. Tracks can show static and dynamic sequence diagrams of an assembly program. Currently Tracks is based on a custom XML format that represents execution traces. While this format can be produced by any suitable debugger, Tracks was developed to work with IDA Pro (and won the Hex-Rays 2011 plugin contests [14]). Thus. to use Tracks with our simulator, we have to generate XML. Figure 7 illustrates the resulting sequence diagram of simulating the REIL code.

Unfortunately, not all REIL instructions produce the control flow information required by Tracks. As such, the implementation of specific REIL instructions inside the simulator will trigger functionality embedded in the simulator's sub-modules to generate the instruction-specific XML data. Control flow information is currently limited to capturing the control flow changes effected by the `jcc` instruction. Specific to this case is determining whether control flow changes as a result of the `jcc` instruction, hence requiring a jump to the beginning of a code block. Secondly, if and only if such a change in control flow is detected will the simulator trigger its XML producing sub-module. The starting address of the block, or the target of the jump is then extracted and propagated to the sub-module, ready to be placed in an XML file. Other instructions can lead to the generation of data flow information or taint information.

When operating alongside IDA Pro, Tracks uses sockets to listen to messages while its debugging view is open to create the trace diagram on the fly, which, once complete, can be saved in this same XML format. This is another mechanism that could be used in place of an XML file to dynamically generate the trace files. Additionally, it is important to note that Tracks was originally designed to trace the flow between subroutines. However there is no bar to using it to trace any flow of information. In our example, we use Tracks to visualize intra-function flow, using jumps to addresses instead of jumps between subroutines. This is due to the fact

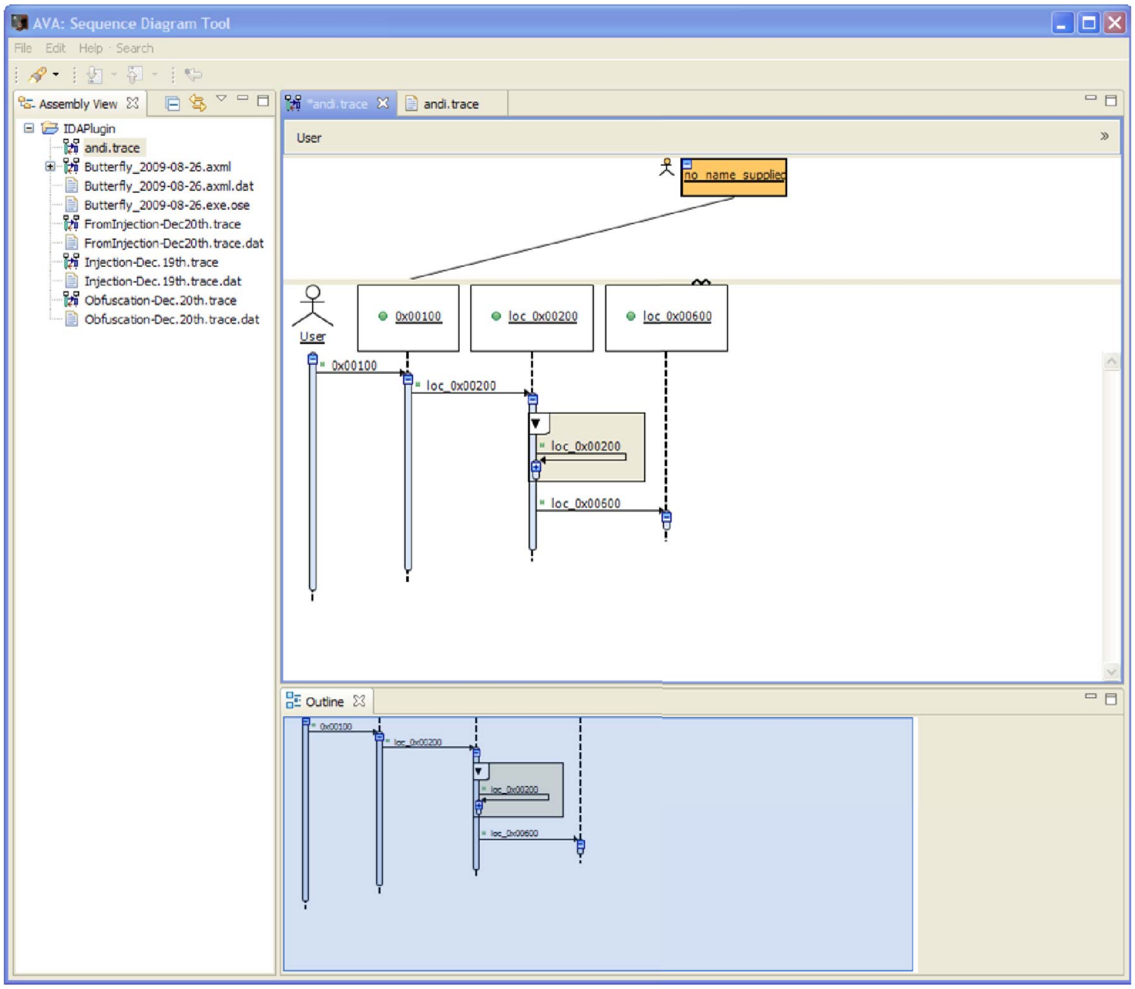

AVA: Sequence Diagram Tool
File Edit Help Search
Assembly View
IDAPlugin
andi.trace
Butterfly_2009-08-26.axml
Butterfly_2009-08-26.axml.dat
Butterfly_2009-08-26.exe.ose
FromInjection-Dec20th.trace
FromInjection-Dec20th.trace.dat
Injection-Dec.19th.trace
Injection-Dec.19th.trace.dat
Obfuscation-Dec.20th.trace
Obfuscation-Dec.20th.trace.dat
*andi.trace
andi.trace
User
no_name_supplied
User
0x00100
loc_0x00200
loc_0x00600
0x00100
loc_0x00200
loc_0x00200
loc_0x00600
Outline

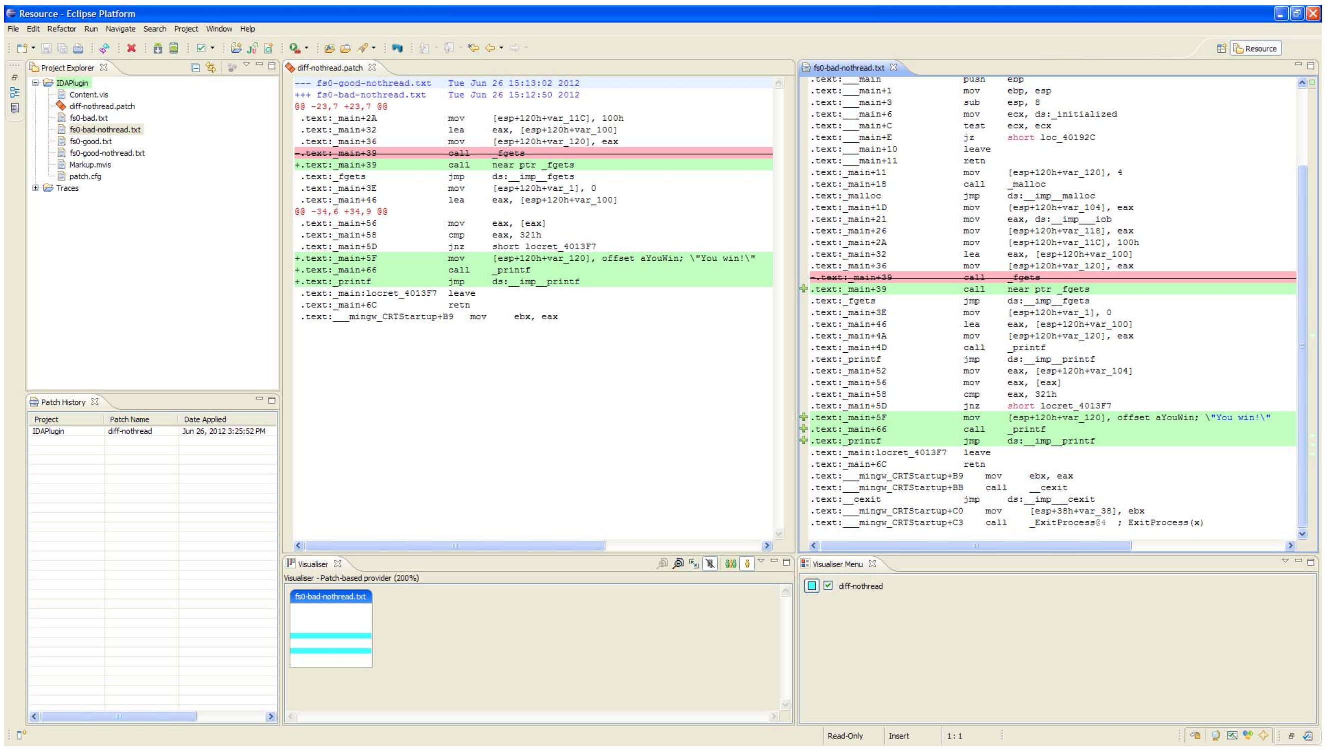
Figure 7. Tracks output of a small string length REIL implementation

that REIL does not support subroutines directly and further extension (through either the language or API) are required to make this possible. With this information, we plan to extend Tracks to allow navigation to finer granularity of control flow—from subroutines, down to basic blocks down to the address level.

To revisit Figure 7, what we see is that the user initiated the execution of the program and our entry point is at address `0x00100`. The next jump is to address `0x00200`. There are then four jumps back to itself (displayed as an expandable loop within Tracks), this is due to the character pointer being moved forward. Finally there is a jump to address `0x00600`. These addresses are prefixed with `loc_`, standing for location, a foible of IDA Pro. This does not imply a subroutine at this address (prefixed by `sub_` in IDA Pro). Additionally, the top portion of the trace file contains a box with the value `no_name_supplied`. These boxes would indicate where the subroutines would be. For example, a call to `printf` would reside in an external library.

*3) Systems tool support for dynamic traces:* Eclippers [15] was created as a part of Integrated System Infrastructure Support (ISIS) and provides patch functionality through Eclipse plugins. The features Eclippers provides are the ability to view changes either already applied, or that will apply, inline within the code. It also provides a history view to show one patch at a time, as well as a system-wide view of affected files using the Visualizer that was originally part of AJDT [16]. Using the refresh button in the top toolbar, all patches can be shown at once (which will also alert the user to conflicts). The 'X' button next to it will turn off all markers and highlighting.

To demonstrate how a tool such as Eclippers can be used in the current context, we created two trace files with the trace replayer available in the latest version of IDA Pro (6.3). The traced program simply echoes back a string that the user enters. However if you give the program the string "%08x%08x%08x%08x%.500u%.269u%n" the pro-



| Project | Patch Name | Date Applied |
|---|---|---|
| IDAPlugin | diff-nothread | Jun 26, 2012 3:25:52 PM |

```
--- fs0-good-nothread.txt   Tue Jun 26 15:13:02 2012
+++ fs0-bad-nothread.txt    Tue Jun 26 15:12:50 2012
@@ -23,7 +23,7 @@
 .text:_main+2A            mov     [esp+120h+var_11C], 100h
 .text:_main+32            lea     eax, [esp+120h+var_100]
 .text:_main+36            mov     [esp+120h+var_120], eax
~~-.text:_main+39            call    _fgets~~
+.text:_main+39            call    near ptr _fgets
 .text:_fgets              jmp     ds:__imp__fgets
 .text:_main+3E            mov     [esp+120h+var_1], 0
 .text:_main+46            lea     eax, [esp+120h+var_100]
@@ -34,6 +34,9 @@
 .text:_main+56            mov     eax, [eax]
 .text:_main+58            cmp     eax, 321h
 .text:_main+5D            jnz     short locret_4013F7
+.text:_main+5F            mov     [esp+120h+var_120], offset aYouWin; \"You win!\"
+.text:_main+66            call    _printf
+.text:_printf             jmp     ds:__imp__printf
 .text:_main:locret_4013F7  leave
 .text:_main+6C            retn
 .text:___mingw_CRTStartup+B9   mov     ebx, eax
```

```
.text:___main             push    ebp
.text:___main+1           mov     ebp, esp
.text:___main+3           sub     esp, 8
.text:___main+6           mov     ecx, ds:_initialized
.text:___main+C           test    ecx, ecx
.text:___main+E           jz      short loc_40192C
.text:___main+10          leave
.text:___main+11          retn
.text:_main+11            mov     [esp+120h+var_120], 4
.text:_main+18            call    _malloc
.text:_malloc             jmp     ds:__imp__malloc
.text:_main+1D            mov     [esp+120h+var_104], eax
.text:_main+21            mov     eax, ds:__imp___iob
.text:_main+26            mov     [esp+120h+var_118], eax
.text:_main+2A            mov     [esp+120h+var_11C], 100h
.text:_main+32            lea     eax, [esp+120h+var_100]
.text:_main+36            mov     [esp+120h+var_120], eax
~~-.text:_main+39           call    _fgets~~
.text:_main+39            call    near ptr _fgets
.text:_fgets              jmp     ds:__imp__fgets
.text:_main+3E            mov     [esp+120h+var_1], 0
.text:_main+46            lea     eax, [esp+120h+var_100]
.text:_main+4A            mov     [esp+120h+var_120], eax
.text:_main+4D            call    _printf
.text:_printf             jmp     ds:__imp__printf
.text:_main+52            mov     eax, [esp+120h+var_104]
.text:_main+56            mov     eax, [eax]
.text:_main+58            cmp     eax, 321h
.text:_main+5D            jnz     short locret_4013F7
.text:_main+5F            mov     [esp+120h+var_120], offset aYouWin; \"You win!\"
.text:_main+66            call    _printf
.text:_printf             jmp     ds:__imp__printf
.text:_main:locret_4013F7  leave
.text:_main+6C            retn
.text:___mingw_CRTStartup+B9   mov     ebx, eax
.text:___mingw_CRTStartup+BB   call    __cexit
.text:__cexit             jmp     ds:__imp___cexit
.text:___mingw_CRTStartup+C0   mov     [esp+38h+var_38], ebx
.text:___mingw_CRTStartup+C3   call    _ExitProcess@4  ; ExitProcess(x)
```

Figure 8.   Eclippers showing the differences between the good and bad trace files

gram will further tell you "You win!" by overflowing the stack and changing the value of the victory cookie. The trace replayer file contains a thread ID which is irrelevant for the comparison and was therefore removed. By doing a diff between the good and bad traces, we arrive at the visualization shown in Figure 8.

In this figure, we see the patch history view on the lower left corner, underneath the package explorer. The patch editor in the middle shows the differences between the two traces, with the suspicious trace on the right. This view shows the differences inline so they can easily be seen. Below the code views are the Visualiser views showing the files affected (and the lines affected within them) with the *diff-nothread* patch file. Lines that have been added to files are highlighted in green and removed lines are highlighted in red and struck through. When patches have not yet been applied, a gutter annotation is shown that when moused over shows the lines to be added or removed.

We foresee using this support to more intelligently visualize the differences between dynamic traces of a program, as well as visualizing the differences between healthy and potentially infected files.

## III. Parallel Possibilities

Software Development Life Cycle (SDLC) models [17] are arguably structured modern practice. The new era of parallel programming is forcing us to rethink these models. While parallelism is considered to be highly performance focused, the impact of developing and tuning a parallel program are rarely well contained within a single software artefact. Instead, parallel software comprehension must consider multiple perspectives in concert as a change in platform (hardware or OS) can force changes to static artefacts, such a source code and configuration and build files. Subsequent impact to the dynamic output must also be tracked, possibly through profiling infrastructure, which can impact memory footprint, ultimately causing performance overhead. Though current SDLC models support an iterative and structured approach to software development, they currently do not explicitly take into account the inherent complexities of the dynamic consequences associated with parallel development.

We believe the integrated nature of the ICE framework provides analysts with a navigable view of multiple artefacts that impact the performance of a parallel program. With a foundation built on the REIL translator, a detailed view of the intermediate representation makes the disassembled program amenable to manual, static analysis as well as input to the other tools.

While the translation example (Figure 1), given in Section II is only a single function, static inspection of the entire codebase of a program requires larger scale understanding of how code segments interact. The visualization provided by MapUI provides the higher level view of this interaction between the parts of the program. Navigation between the high-level map perspective and the detailed REIL code provides an analyst with the ability to decipher the details of a single function while keeping it in context with the other pieces of the program it interacts with.

While the static view of a program provides an analyst with some idea of the program's behaviour, the dynamic result of a program's execution must almost always be taken into consideration to get a complete picture. ICE's interface with the simulator (Subsection II-B) enables a perspective shift from static to a detailed listing or abstract representation of the dynamic results generated by the simulator. The raw output of the simulator allows an analyst to inspect the detailed use of registers and memory with respect to the line-by-line, static code perspective. The simulator results feeding into Tracks provide a visual representation of the program's control flow. Again, navigation between the static code representation and the dynamic execution of the system can help in the identification of unexpected program behaviour.

## A. *Extension for Parallel Instructions*

The x86 instruction set has been extended several times in the past to accommodate extra functionality and expanded registers. Intermediate languages have served to abstract such complexity and to shield developers while remaining amenable to analysis tools. As an example, Figure 9 provides an alternative implementation of the string length function that uses Single instruction, multiple data (SIMD) instructions to reap the rudimentary parallel benefits of the underlying hardware.

The code sample in Figure 9 contains several SIMD instructions: `pxor`, `movdqa`, `pcmpeqb` and `pmovmskb`. Some of these are a scalar version of a single instruction, single data (SISD) instruction and as such can be mapped to it. The `pxor` instruction can be directly mapped to the `xor` instruction. Likewise, `movdqa` can map to a standard move instruction, which in REIL would correspond to `str` for moving between registers and `stm` or `ldm` for memory manipulations. Dealing with the original requirement for memory alignment on 16-byte boundaries of `movdqa` cannot be directly effected in REIL, since exceptions are processor specific.

Both `pcmpeqb` and `pmovmskb` require special attention. Due to the nature of these instructions, a sequential breakdown of the translation may be deemed beneficial since no direct SISD and REIL conversions exist. Alternatively, extending the REIL instruction set in such cases to allow the simulator correctly deal with SIMD instructions should be considered.

The design and implementation of ICE, ranging from its pluggable interface to the extensibility of the chosen intermediate language makes it amenable to the analysis of SIMD instructions. In one approach, the extension would occur at the level of the x86 to REIL translator within ICE.

```
; get pointer to string
mov      eax,  [esp+4]
  ; copy pointer
  mov      ecx,  eax
  ; set to zero
  pxor     xmm0, xmm0
  ; lower 4 bits indicate misalignment
  and      ecx,  0FH
  ; align pointer by 16
  and      eax,  -10H
  ; read from nearest preceding boundary
  movdqa   xmm1, [eax]
  ; compare 16 bytes with zero
  pcmpeqb  xmm1, xmm0
  ; get one bit for each byte result
  pmovmskb edx,  xmm1
  ; shift out false bits
  shr      edx,  cl
  ; shift back again
  shl      edx,  cl
  ; find first 1-bit
  bsf      edx,  edx
  ; found
  jnz      A200

; Main loop, search 16 bytes at a time
  ; increment pointer by 16
  A100:   add      eax,  10H
  ; read 16 bytes aligned
  movdqa   xmm1, [eax]
; compare 16 bytes with zero
  pcmpeqb  xmm1, xmm0
; get one bit for each byte result
  pmovmskb edx,  xmm1
; find first 1-bit
  bsf      edx,  edx
; loop if not found
  jz       A100

  ; Zero-byte found. Compute string length
  A200:
  ; subtract start address
  sub      eax,  [esp+4]
  ; add byte index
  add      eax,  edx
  ret
```

Figure 9. Sample x86 code for string length with SIMD instructions

The translator would take a parallel instruction, like the SIMD instructions here, and sequentialize it and translate the sequential equivalent into existing REIL instructions. The benefits of this approach is the ability to leave REIL and existing analysis tools intact and confine changes strictly to the ICE translator. The majority of SIMD instructions can be converted without much difficulty into a sequential block of several SISD instructions. The challenges are related to both the complexity of the SIMD instructions and the vast number of possible flag settings. That is, the sequential implementation of each instruction is difficult to identify and the side effects of each instruction can alter the intended simulated program execution. Additionally, the flag settings that can occur in a single SIMD instruction may be very different than its equivalent sequential counterparts. The occurrence of this is not preventable in all cases and, as such, must be avoided within analysis. Consider that a SIMD instruction set flags in a deterministic way. That is, one knows ahead of time which flags may be set, unset, left undefined, or not touched at all.

REIL itself does not support any SIMD instructions, but given the simplicity of the instruction set and its extensibility, another option is to extend the current set of REIL instructions to include parallel instructions. The benefit of this approach is that the analysis would be parallel-specific and would be controlled both at the level of the framework and the tools, providing more accuracy. The down side of this approach is the loss of REIL's simple instruction set. In the simplest form, a one-to-one mapping would add a new REIL instruction for every SIMD instruction. Considering Intel's Streaming SIMD Extensions 2 (SSE2) instructions alone, this would add upwards of 100 new REIL instructions to the existing set. To extend the size of the REIL instruction set in this manner would add complexity to the new instruction definition and, additionally, each plugin would have to be extended to handle these parallel REIL instructions.

## IV. Discussion and Conclusion

Reverse engineering is hard. Although there have been great strides in assembly analysis tools in the last few years, the vast majority of these tools have been developed exclusively for x86 assembly and single-thread execution. Thus, there are still many problems to be solved to provide a complete solution, even for x86.

Unfortunately, with the advent of new platforms such as iOS and Android, the age-old problem of understanding assembly code for vulnerability and malware analysis is starting anew. Even more unfortunately, nearly all of the great existing tools need to be reinvented for these new platforms.

For this reason, we propose ICE, an Integrated Comprehension Environment that will unhinge a platform's instruction set from the analysis and visualization modules. Hopefully, this will allow new tools to work on more than one platform, benefit from parallelization (both from the analysis' standpoint and for tool efficiency) and increase the speed at which we can respond to ever increasing cyber threats!

## References

[1] I. Guilfanov. (2006) Automated binary analysis woes. [Online]. Available: http://www.hexblog.com/?p=43

[2] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, "Codesurfer/x86a platform for analyzing x86 executables," in *Compiler Construction*. Springer, 2005, pp. 139–139.

[3] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," *Information Systems Security*, pp. 1–25, 2008.

[4] J. Kinder, "Static analysis of x86 executables," Ph.D. dissertation, Technische Universität Darmstadt, 2010.

[5] T. Dullien and S. Porst, "Reil: A platform-independent intermediate representation of disassembled code for static code analysis," *CanSecWest*, 2009.

[6] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[7] J. Baldwin, D. Myers, M.-A. Storey, and Y. Coady, "Assembly Visualization and Analysis. An Old Dog CAN Learn New Tricks!" in *Proceedings of the 2009 OOPSLA Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, Orlando, FL, 2009.

[8] J. Baldwin, P. Sinha, M. Salois, and Y. Coady, "Progressive user interfaces for regressive analysis: Making tracks with large, low-level systems," in *Australasian User Interface Conference (AUIC 2011)*, vol. 117, Perth, Australia, 2011, pp. 47–56.

[9] J. Baldwin and Y. Coady, "Social security: collaborative documentation for malware analysis," in *Proceedings of the 12th Annual Conference of the New Zealand Chapter of the ACM Special Interest Group on Computer-Human Interaction*, ser. CHINZ '11. New York, NY, USA: ACM, 2011, pp. 17–24. [Online]. Available: http://doi.acm.org/10.1145/2000756.2000759

[10] "High Level Assembler and Toolkit Feature," 2010, http://www-01.ibm.com/software/awdtools/hlasm.

[11] D. Pucsek, J. Wall, C. Gibbs, J. Baldwin, M. Salois, and Y. Coady, "Ice: circumventing meltdown with an advanced binary analysis framework," in *proceedings of the First International Workshop on Developing Tools as Plug-ins (TOPI)-colocated with the International Conference on Software Engineering (ICSE)*, 2011.

[12] R. DeLine, "Staying oriented with software terrain maps," in *DMS*, A. Guercio and T. Arndt, Eds. Knowledge Systems Institute, 2005, pp. 309–314.

[13] Crackmes.de, "reversers' playground," 2012, last visited June 2012. [Online]. Available: http://crackmes.de

[14] Hex-Rays, "Plug-In Contest 2011: Hall of Fame," 2011, last visited June 2012. [Online]. Available: http://www.hex-rays.com/contests/2011/index.shtml

[15] J. Baldwin and Y. Coady, "Adaptive systems require adaptive support–when tools attack!" *Hawaii International Conference on System Sciences*, vol. 0, p. 259, 2007.

[16] The Eclipse Foundation, "The Visualiser," 2010. [Online]. Available: http://www.eclipse.org/ajdt/visualiser

[17] P. Kruchten, *The Rational Unified Process: An Introduction*, 3rd ed. Boston: Addison-Wesley, 2003.