Id - Dynamic Views on Static and Dynamic Disassembly Listings

Nicholas Sherlock
Computer Science
University of Otago
Otago 9010 New Zealand
n.sherlock@gmail.com

Andrew Trotman
Computer Science
University of Otago
Otago 9010 New Zealand
andrew@cs.otago.ac.nz

Abstract *Disassemblers are tools which allow software developers and researchers to analyse the machine code of computer programs. Typical disassemblers convert a compiled program into a static disassembly document which lists the machine instructions of the program. Information which would indicate the purpose of routines, such as comments and symbol names, are not present in the compiled program. Researchers must hand-annotate the disassembly in a text editor to record their findings about the purpose of the code.*

Although running programs can change their layout dynamically, the disassembly can only show a snapshot of a program's layout. If a different view of a program is required, the document must be recreated from scratch, making it difficult to preserve user annotations.

In this paper we demonstrate a system which allows a disassembly listing to be refined by user input while retaining user annotations. Users are able to dynamically change the interpretation of the layout of the program in order to effectively analyse programs which can alter their own memory layout. We allow users to combine the independent analysis of several program modules in order to examine the interaction between modules.

By exploring the obsolete "Poly" computer system, we demonstrate that our disassembler can be used to reconstruct and document entire software distributions.

Keywords Digital Libraries, Cognitive Aspects of Documents, Document Workflow

1 Introduction

The rate of computer hardware and software development is increasing exponentially. Five years ago, our desktop computers were all powered by single-core CPUs. Two years ago, they had dual-core CPUs. And today, they are likely to have four or eight cores. The Macintosh series of computers have seen large architectural changes, switching from Motorola CPUs to PowerPCs and finally to Intel x86 CPUs. In successive steps, we have changed our removable storage media from tapes, to 8, 5¼ and 3½ inch floppy disks, to CD-ROMs, DVDs, Blu-ray, and increasingly, removable flash-memory based storage. With each new hardware generation, our old software becomes obsolete and is either rebuilt or abandoned.

This creates a problem for researchers and historians. While design manuals can be scanned and stored accessibly in a digital library, and data can be retrieved from old media with somewhat more effort and expense, storing the software from these old machines in a useful format is an entirely different problem. Performing analysis on software which is stored in the library becomes increasingly difficult with time. This is because a piece of software cannot be used, examined, or understood in isolation. Its behaviour is defined by its interaction with the hardware it was built for. Obsolete hardware becomes progressively more scarce with time. Preserving old hardware by building modern replicas requires an increasingly infeasible amount of effort and resources as microelectronics become more complex.

If an accurate description of the hardware is provided or can be discovered, it can be replaced by a software-based "emulator". An emulator in this context is a program which simulates the action of an old hardware platform on (typically many) modern platforms. In this way, researchers can examine the runtime behaviour of old software without having to perform a costly hardware reconstruction of an old platform.

A second problem is the difficulty of examining the algorithms and implementation details of obsolete software when human-readable source code has been lost or was never provided. It is also a problem for modern software. For example, in order to build a new program which interoperates with an existing program, some knowledge of the original program's internal operation is required. Even if the source code for a program is available, you may still want to examine the machine instructions that the compiler generates to ensure that the generated instruction sequences are correct or efficient. Machine code is a more primitive level of abstraction which can reveal surprising negative performance implications of innocuous-looking high-level code.

If only the machine code that makes up the compiled program is available, it must first be translated

Proceedings of the 14th Australasian Document Computing Symposium, Sydney, Australia, 4 December 2009.

into a more abstract form that humans can understand so that it can be analysed. A program which performs this translation is called a "disassembler". Much of the information found in the high-level source code of a program, including comments and the names of variables and routines, is lost in the compilation process. To understand a compiled program, a researcher must recover this lost information. They can achieve this by inspecting the disassembly listing with reference to the behaviour of the running program. They can then share their findings with other researchers by annotating the disassembly.

Several factors make this process difficult. The disassembler's interpretation of the program must be dynamically altered to analyse programs which can change their layout at runtime. In order to change the interpretation of the program, the disassembler must be re-run. This creates a new, independent disassembly document, which makes it difficult to preserve annotations that the researcher has already made.

Even if a program does not change its layout dynamically, the researcher must still frequently change the disassembler's interpretation of the program. This is because the disassembler cannot distinguish code from data in the compiled program with perfect accuracy. Human judgements are required to correct the disassembler's mistakes.

In order to allow researchers to effectively document entire programs, software support is required to assist the user in navigating and imposing structure on large disassembly documents, but this is not provided with a traditional disassembler. Although programs being analysed are often composed of several related modules which can be examined independently, disassemblers typically do not provide any way of linking disassemblies together in order to share information about interacting modules.

In this paper, we will present our disassembly, debugging and emulation system which we used to reconstruct and document the software and hardware of the "Poly" computer system. We will show that our disassembler can solve the problems inherent in documenting the software of the Poly by using it to create a digital Poly software library which researchers will be able to examine long into the future.

2 The Poly computer system

The Poly was a computer system developed in New Zealand in the early 1980s. It was comprised of a server computer called the "Proteus" with a series of fat-client "Poly" machines attached by a token ring network. It was designed to be used in a classroom setting where a teacher would set work on the server computer to be distributed to each student's computer. When the students finished their work, their results would be sent back to the server computer to be saved to disk. The server and the client machines had similar architectures. In one prototype, a client could be turned into a Proteus server with the addition of a disk drive. The computers can be seen in Figure 1.

The Poly never gained much ground in the computer market and few machines were produced. Although the Poly demonstrated innovative technologies and ideas, and is an important part of New Zealand's computing history, little is now known about it. In particular, the Poly's networking capabilities were far ahead of contemporary computers, and it was provided with innovative classroom software to take advantage of those features. But with only a couple of working Polys in existence and little surviving documentation, the exact functionality of the software is largely a mystery.

In order to make the Poly's software available to researchers, we would have to document it in a form that would be useful long after the last Poly stops operating.

3 Disassembly

```
48A6  34 14        PSHS X,B ;Ref from $CD39
48A8  8E 5B 19     LDX #$5B19
48AB  C6 05        LDB #$5
48AD  E7 80        STB ,X+
48AF  35 04        PULS B
48B1  E7 80        STB ,X+
48B3  35 20        PULS Y
48B5  EC A4        LDD ,Y
48B7  ED 84        STD ,X
48B9  8E 5B 19     LDX #$5B19
48BC  10 8E 00 04  LDY #$04
```

Figure 2: A fragment of a disassembly listing

A tool called a "disassembler" examines a program binary (that is, the machine code that the computer will execute, not the source code which is used to generate it) and creates a text file called a disassembly listing. A disassembly listing shows the machine code instruction that appears at each memory address within the program as a human-readable mnemonic code. It also shows the data stored inside the program, such as the text of string literals or numeric literals from the source code.

Figure 2 shows a fragment of a program disassembly for the Poly's Motorola 6809 CPU[8]. The leftmost element is the memory address of the disassembled instruction. Next is a hexadecimal representation of the machine code that the CPU will execute. Finally, a human-readable interpretation of the machine code is displayed. The first part of the instruction is a mnemonic which represents the instruction being performed (for example, PSHS is an instruction to push a value onto the stack). Any arguments to the instruction follow the mnemonic. X, B and other symbols refer to registers on the CPU and values starting with a hash symbol are numeric literals. There is effectively a one-to-one mapping between the machine code and the mnemonic representation shown to the researcher.

(a) Two Poly client machines sit side-by-side

(b) A Proteus server and its CPU and memory board (inset)

Figure 1: The key components of the Poly system

There are two major difficulties in building a useful disassembler program. The primary difficulty is that it is impossible in general to automatically decide which parts of the program binary are data and which parts are code which will be executed. This problem is equivalent to the halting problem[5]. Because of this, disassemblers must sometimes guess where a machine instruction begins in memory and so will make some incorrect guesses. Wrong guesses might identify the beginning of a sequence of instructions at the wrong offset (so that the interpretation of the sequence begins halfway through a machine instruction, generating incorrect output,) or incorrectly identify data as code or vice versa, which hampers correct interpretation of the program. Some code locations can not be identified because their addresses are computed at runtime by the program in a way that the disassembler cannot predict. For example, a program may read the address of the routine to execute from an external file.

The second difficulty is encountered when analysing software that was built for small systems like the Poly. Like many computers of its time, the Poly had more physical memory available than it could simultaneously address. Its CPU's memory address bus is 16-bits wide, allowing it to address 64kB of virtual memory at any one time. The Poly has 128kB of physical RAM plus 8kB of BIOS and memory-mapped peripherals. Software on the Poly dynamically changes the mapping of the 8kB virtual memory pages to the 128 + 8kB physical address space by changing the entries in a memory map.

In Figure 3, a 16-bit virtual memory address is translated into a 17-bit address in physical memory in a series of steps. In "protected" mode (operating system mode), some addresses are directed to hardware and the BIOS. Otherwise, the three most-significant bits of the virtual address are combined with a bank-select bit and used as an index into the programmable memory map. The memory map replaces the three higher bits of the virtual address with four bits of its own, creating a 17-bit address in physical memory.

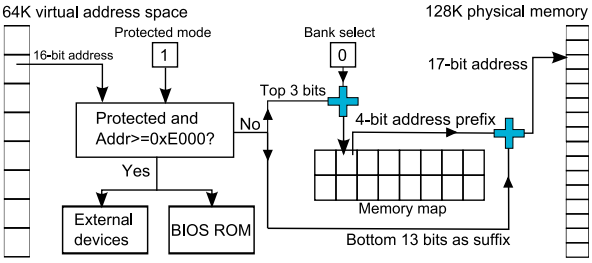


Figure 3: A virtual memory address is translated using the memory map into a physical address

With a traditional disassembler, the researcher would have to disassemble the program once for every memory mapping they wanted to examine, and maintain the different disassembly listings independently, even when information should be shared between them. The same physical memory page can even appear in virtual memory in more than one place simultaneously, making it difficult to manually keep annotations consistent and up to date.

Many small-CPU based systems, including systems of about the Poly's age, use dynamic memory maps to overcome the limitations of restrictively small address spaces. For example, the Apple //e[9], ZX Spectrum 128[1] and Commodore 128[6], which, like the Poly, have 16-bit address busses and can support 128kB or more of memory. Software written for 16-bit operating systems such as MS-DOS on more modern PCs or software for embedded systems also use this technique. To effectively analyse these systems, a new kind of disassembler is required.

4 Background

Disassemblers are available for nearly every platform. Disassembly tools are available in two main contexts: As a static disassembler tool to examine stored programs on disk, or dynamic disassemblers which examine snapshots of running programs.

4.1 Static analysis

The tool "objdump"[2] typifies static disassembly tools. A binary program on disk is provided as input to `objdump`, and the output is a disassembly listing document.

The generated listing may be explored and annotated with a simple text editor, but this approach has two serious disadvantages. Firstly, a text editor treats the disassembly as unstructured text and so can offer very little software support for common annotation operations. It will not offer cross-referencing support, so any references that the code makes to other parts of the program must be followed manually. If the analyst gives a descriptive label to a block of code, that label will not be propagated to the places where the code is called. The analyst cannot effectively experiment with different interpretations for data stored in the program. For example, to reinterpret a number as signed or unsigned will require the researcher to manually convert the number using some other tool, or re-run the disassembler to create an entirely new disassembly. These problems dramatically slow down analysis and make understanding the program much more difficult.

Secondly, a static disassembler cannot always correctly distinguish code from data in the analysed program. For example, the code may include a jump whose target is an address which is computed at runtime. This is common in object-oriented code, where the address of a virtual method must be looked up in an object's virtual address table. In procedural code, this technique is more likely to be used with a jump table—a table of routine addresses that selected from at runtime, often by an equivalent of the "switch" statement in C. Data-flow analysis techniques could be used to discover the targets for some of these computed jumps[4]. For instance, the instruction sequence `LDX #0x5B19 / JMP X` (storing the value `0x5B19` into the register `X`, followed by a jump to the value stored in `X`) is clearly a jump to the location `0x5B19`. Even so, some jumps are computed in a way that no disassembler could possibly understand (e.g. by using data available at runtime which is not present in the image being disassembled, such as data contained in a message received on the network.)

The disassembler might identify a jump with a known target which in fact points to data, not code. If the Poly jumped to that location, it would likely have unexpected results, perhaps crashing. If we assume that the the Poly code does not crash, it is reasonable to assume that it does not take bad jumps. There are at least two possible causes of this situation.

It may be impossible for the flow of execution to ever reach the jump, so the bad jump is never executed in practice. For instance, a program might check the state of a "debugging mode flag", and, based on the value it finds, jump to some logging routine which ended up being cut from the final binary. The debugging mode flag is never set in delivered software so the bad jump is never taken.

The jump may have a definite target, and be taken at runtime, but the target of the jump which is stored in the instruction is overwritten at runtime before the jump is ever called. This is seen on modern architectures. A module of code (such as a Windows dynamic-link library or a Unix shared object) which a program uses may be dynamically loaded at an unpredictable position in its address space. To be able to call routines from the module, the program needs to know their addresses. To achieve this, an "import table" is generated in the application. The import table consists of a series of stubs. The stubs are small routines which contain a jump to an address which is initially some default value (`NULL`). When the library is loaded, the memory locations of its routines are discovered and used to rewrite the code in the import table. To call an imported routine from within the program, a call to the stub is made some time after the library is loaded. Calling the routine before the library is loaded results in undefined behaviour.

In order to correct code which has been misidentified as data, or vice-versa, the user must run the disassembler again with that new information. This produces an entirely independent disassembly listing which must then be manually merged with the listing the user has annotated. This is an error-prone and tedious process. The user is unlikely to want to experiment with different interpretations of a memory address, because each experiment is so costly to run in terms of user effort.

4.2 Dynamic analysis

A debugger like the free tool "gdb"[7] is designed to allow the user to inspect and interact with running programs. If no debugging information or source code is provided which would allow it to show the high-level code that corresponds to the running machine code, it uses an embedded disassembler to show the disassembly of the code that is currently executing.

This approach has several advantages. Code can be distinguished from data with certainty, since the debugger only needs to show the disassembly for instructions which are currently executing or have previously executed. The user can have the debugger interpret any memory location in multiple ways. For example, they could view one location as both an array of integers and an array of characters, and discover that the data only makes sense when interpreted as an array of integers.

The analyst can interact with the running program to see what inputs a piece of code receives, or precisely what action it takes as a result. The running program may be modified by the analyst to explore areas of code that would not normally execute. For example, they can force the code to follow an error-handling branch in order to examine that mechanism, even if they do not know what inputs to the program are needed to cause the error to be triggered in normal execution.

The main disadvantage of this approach is that the user is typically unable to add any annotations to the disassembly. If they discover the purpose of a routine,

they cannot give it a human-readable label which would allow it to be understood the next time it is encountered. Even if annotations are supported, the debugger will not provide any way to save them and load them again later, since it has no expectation that the memory layout of the program will be similar the second time the program is run. Analysis with a debugger is ephemeral, it cannot be effectively used to produce a document which could record the user's findings to be shared with other researchers.

4.3 Interactive disassembly

Traditional disassemblers are frequently used in situations where the disassembly is only useful for a short amount of time, like a single session, and saving annotations is less important. For example, a common task for a traditional disassembler is examining the machine code generated by a procedure in a high-level language to diagnose performance or code generation issues. Since they are typically used to examine a program which is currently in development (and therefore changing dramatically from a machine code perspective), the ability to save annotations is not valuable.

If a disassembly listing is to be modified and examined over an extended period of time (i.e. several analysis sessions), or shared with other people, it must be able to change dynamically as more information is discovered by a human researcher. The researcher will work *with* the disassembler to analyse a program. This is the approach that we decided to take with our own disassembler.

The only interactive disassembler that we are aware of in common usage is IDA[3]. But IDA does not support the dynamic memory model of the Poly. While it supports debugging live code for some targets, it does not integrate with our Poly emulator.

5 Id, the interactive disassembler

To assist our reconstruction of the Poly platform, we developed "Id", an interactive disassembler which supports the Poly's CPU and binary layouts. Id is an application for Windows with a Graphical User Interface. The main pane of Id is the disassembly listing. Surrounding the listing are panels that give extra information about the binary being disassembled. For instance, one panel is a list of all of the symbol names created so far in the image. Id can be seen in Figure 4.

To support the changing layout of programs on the Poly, all of the items that Id identifies in its disassembly are tagged with the physical address that they are stored at, not the virtual memory addresses that they appear at with one possible memory mapping. This allows the user to change the virtual memory map while they are examining the disassembly, and have the disassembly listing change dynamically to reflect the new interpretation of the program's layout. This approach works well with the Poly because code and resources on this platform typically have a fixed location in physical memory. On systems with address spaces larger than the amount of physical memory available, like modern 32- and 64-bit computers, the reverse tends to be true. Programs move around in physical memory but stay in a fixed position in virtual memory.

Initially, no code has been identified in the image (it is all considered to be data). To begin the disassembly process, you must identify the start of a machine instruction in the image. The CPU has to perform the exact same task when the Poly boots. The CPU begins by reading an address from an interrupt table at a fixed location in memory, which is called the "reset vector". This is the location where execution begins after boot. Id begins disassembly at this point. If the instruction at the reset vector is a jump to a different location, Id can follow the jump and recursively identify code there. If the instruction is not a jump, execution will continue to the next location in memory, so Id identifies an instruction there. Following jumps from the entry points defined in the interrupt table identifies much of the code in the image—around 60% for the Poly's BIOS. The remainder of the code is often interrupt handlers whose address is determined at runtime in a fashion that is currently too difficult for Id to discover.

To assist Id, the user can create new entry points (the beginnings of instruction sequences) at any time. Id automatically adds disassembly for those previously-unidentified locations to the disassembly listing. If the disassembler has wrongly identified data as code, the user can convert it back to data.

While the physical representation of the data in the program is known from the disassembly, the information that the data encodes cannot always be inferred automatically. For example, Id knows that the operands of instructions which specify the targets of read or writes to memory locations are memory addresses. It can use syntax colouring to distinguish these addresses from other numeric literals found in the program. When possible, the symbolic name that the user has given to the target address is shown in place of the raw address.

However, operands to instructions which are merely stored into registers or into memory locations have no special meaning defined by the instruction set. Id allows the user to experiment with different interpretations of the data in order to discover what information the data is encoding. For example, Id can interpret data as strings, arrays, addresses, or numeric types of various sizes and formats. As even simple operations such as adding a constant to a number stored in a register can have multiple reasonable meanings, this user-directed assistance is crucial to documenting the purpose of the program. For example, the machine code and effect of subtracting 16 from a number stored in a register is identical to that of adding 65520 (`0xFFF0` in hexadecimal notation). But these two operations suggest very different purposes for the code being disassembled. In the first case, the program may be accessing data that appears immediately before a previously-computed ad-

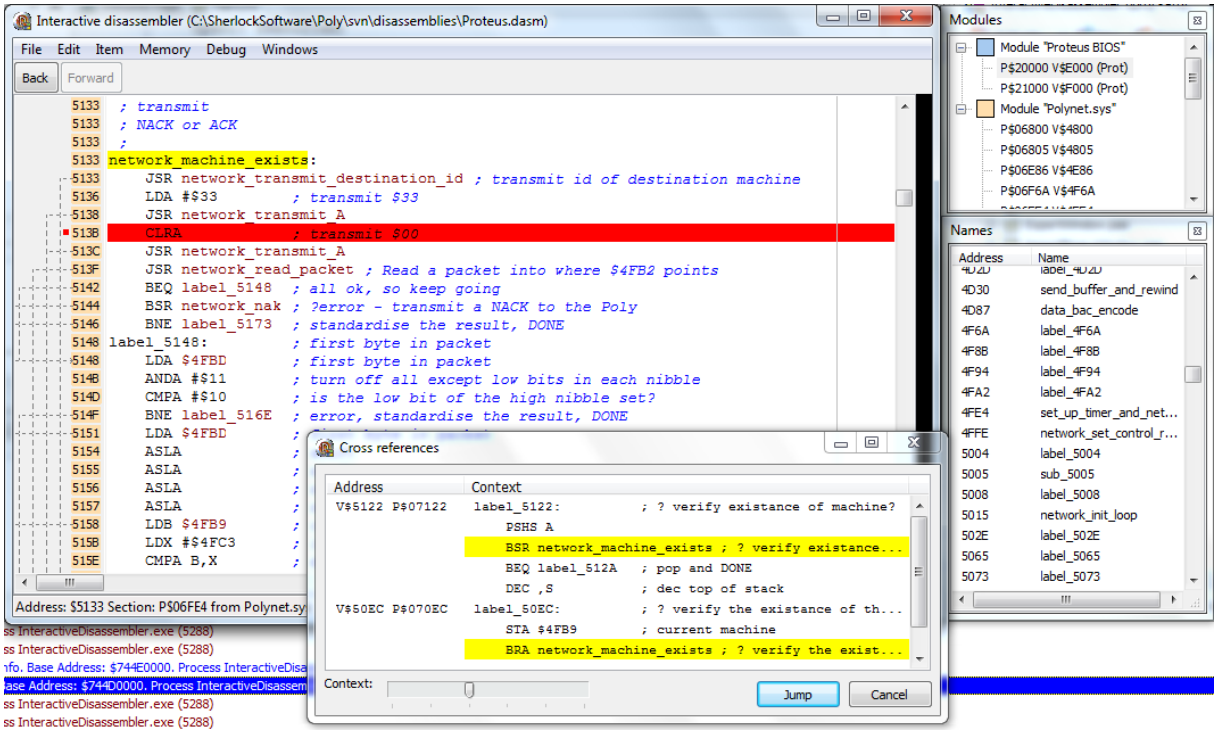


Figure 4: Id's main GUI with the Proteus operating system loaded for disassembly. Andrew Trotman's comments appear after semicolons. All names in the disassembly are user-entered.

dress. This is the pattern expected if the program is iterating over an array of elements whose size is 16-bytes in reverse order. In the second case, the code may be calculating the location of a dynamically-determined field of a structure which is located at the fixed address `0xFFF0`. This is the pattern expected if the program is looking up the current location of one of the installed interrupt routines (as the interrupt-vector table is located at `0xFFF0`). By specifying the signedness of the operand, the user can disambiguate these two cases.

5.1 Annotation

Id is a hypertext document system. The start of each routine or piece of data can be given a title by the user. Id then provides a "names" pane, which lists every title in the program, sorted by module. This structured outline allows the document to be navigated rapidly.

References to memory locations found in the code, such as the targets of jumps or the targets of memory reading instructions, are shown as hyperlinks. The user can double click on the hyperlink to jump directly to its target. If there is a user-supplied title at the target location, it is shown as the anchor text in the disassembly in preference to the target's raw memory address. The user can further describe the purpose of a location by adding an extended comment to the title. This extended comment becomes the default comment which appears automatically at all referral sites. The user can edit the comment at a referral site in order to record the exact way that the target is being used. For example, the researcher might label a memory location as "num_clients", and provide the extended comment "number of Poly clients currently connected" to clarify the meaning of the location. A comment at a referring site which increments this value might then be customised: "record newly-attached client".

As the user adds labels to locations in the memory map, the propogation of these labels to referring sites makes the meaning of previously unexplored code clearer. The analysis process shows a "jigsaw-like" effect. As with a jigsaw, a series of interlinked pieces meet at common interfaces. As the jigsaw is completed, the possible shape and location of the unplaced pieces is futher and further constrained. This implies that the initial analysis—discovering the large-scale structure of the program—is the most difficult phase, with analysis becoming more rapid as the final "pieces" are placed.

Id's hyperlinks are bidirectional. The user can select a title and discover all of the links which point to it by using Id's "cross-references" window. Id helps the user choose interesting referrers for further analysis by showing some context from each incoming link (the closest few instructions and comments). By examining a routine's referrers, the researcher can determine (by trial and error) what kind of inputs will be provided to the routine and what sort of return value is expected in response. Examining referrers is critical to discovering the purpose of a target routine or memory address.

If the memory map changes, the names of the targets of links in the code are updated accordingly. In protected mode, a call to a routine at the address `0xF030` might send instructions to hardware to print

text to the display. But in unprotected mode, a different routine will reside at that address. It could also be a line-printing routine, but it might first copy the user's text to a buffer area before switching to protected mode to call the BIOS's line-printing routine. Id allows calls to these two routines to be distinguished by allowing them to have different names.

Broken hyperlinks, which are links that point to locations which do not currently exist in the disassembly, are highlighted for the user. The presence of a broken link could indicate that the target has not yet been loaded from disk, or that the memory map is expected to change before the link will be accessed by the program. This highlighting is particularly valuable for discovering the connections between loadable modules of code.

While the user can instruct Id to reinterpret a location of the disassembly, the text of the disassembly listing itself cannot be directly edited by the user. This allows Id to ensure that the disassembly listing always corresponds precisely to the machine code. In fact, the listing could be used to reconstruct the original binary program. Id does allow comments to be added to any line of the disassembly as the purpose of routines are discovered. Once a piece of code has been understood, adding a comment allows the purpose of the sometimes confusing assembly to be shared with other researchers.

5.2 Modules

Disassembly will typically be performed on a "module". A module is a set of code and data which is tightly bound together. For instance, one module might be the Poly's BIOS, which is the code that controls the interaction between the hardware and the software. This code is stored in permanent memory chips on the motherboard and appears in the Poly's virtual address space when it enters "protected mode". Another module might be a boot sector read from a floppy disk. Id understands the format of the Poly's file system and programs, so it can simulate the action of the Poly's program loader and load a program from disk as a module into the correct physical memory locations for analysis.

There are substantial cross-references between modules, so the analysis of one module can be used to understand a different module. For example, the BIOS is responsible for loading the boot sector from a floppy disk, then it transfers control to the boot sector program. By disassembling the BIOS, the entry-point of the boot sector's code can be discovered. In a traditional system, the disassembly document of each module is independent so there is no inter-module linking.

To support the analysis of large systems, Id's module system allows multiple disassemblies which were created independently to be composed to appear as one coherent document. For instance, one disassembly might be the *Proteus's* operating system and BIOS. Another disassembly might be the *Poly's* operating system and BIOS. If you created a disassembly of a text editor program, you could choose to either link to the disassembly of the Poly's operating system (to examine the program's effect on the Poly), or link with the Proteus's operating system (to examine the program's effect on the Proteus.) This linking can be easily changed while you are disassembling. The gutter of the disassembly pane is colour-coded to show the module that a line of code belongs to. Cross-module references are dynamically resolved based on the current selection of modules. New information identified about linked modules is automatically used to update those documents when the parent document is saved. This system allows reuse of disassembly information—the information you discover about the operating system while analysing a text editor program is made available when examining the interaction between a database program and the operating system.

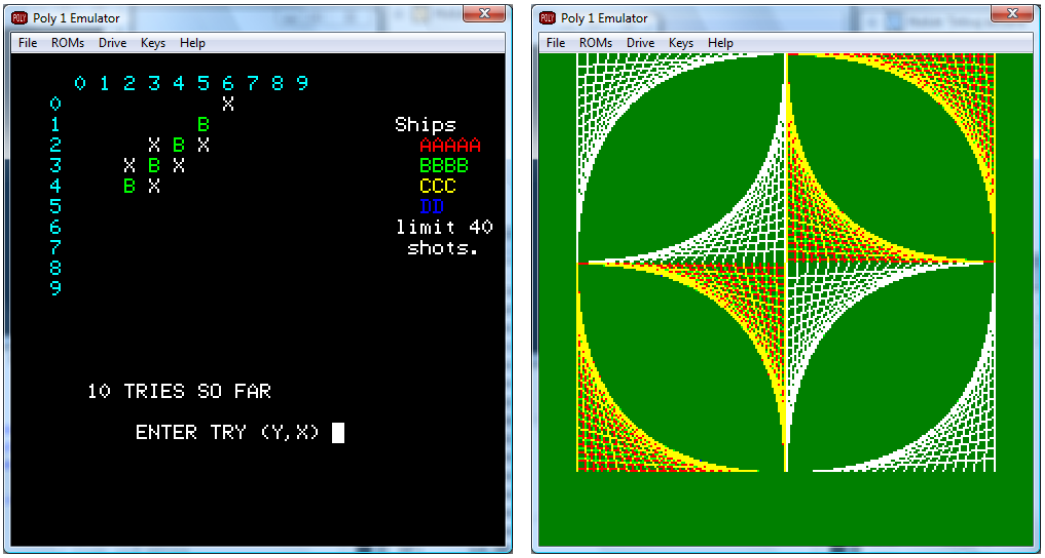
The module system allows the user to document an entire operating system and its attendant application suite as a set of linked disassemblies.

5.3 Debugging

A goal of Id was to unify the capabilities of static and dynamic disassemblers. Id includes a debugger which can attach to a running instance of our Poly emulator. This allows the runtime behaviour of a disassembled program to be examined. It also allows the creation of new disassemblies based on code which is loaded in memory at runtime from unknown sources. For example, the applications on the client Poly machine are loaded from the server across the network. Using the debugger, a copy of a network-loaded application can be dumped from the client for later analysis.

The debugger integrates with the disassembler tightly. For example, the debugger observes the instructions which are executed at runtime in order to discover the location of code in the image which it could not have discovered with a static analysis of the program on disk. The identified code is automatically merged into the document that the user has already created. The user is notified if the newly-identified code is inconsistent with previous disassembly. For example, if a newly-identified instruction lies within a previously-defined instruction (which is vanishingly rare in valid code), the user is asked to decide which interpretation to accept.

By using the debugger, the user can discover cross-references at runtime which cannot be discovered with static analysis. For example, the user can set breakpoints which pause execution when a certain memory location is read to or written from. When the breakpoint is triggered, the user has identified a link which references their memory address. This is particularly useful in debugging the behaviour of hardware devices (which appear at virtual memory addresses in the Poly's memory map).



(a) Emulated text-mode battleships game on the Poly client

(b) Emulated graphics-mode on the Poly client

Figure 5: Emulated Poly machines

In Figure 4, the debugger has been attached to an emulator which is simulating the Proteus server. A breakpoint has been placed in part of a network routine. Execution will automatically pause at that point if the Proteus executes that instruction, allowing the user to inspect the contents of memory and the CPU registers. In Figure 5, two Poly clients with attached debuggers are executing programs delivered by the Proteus server.

6 Conclusions

We used the dynamic document features of our interactive disassembler system, Id, in our analysis and reverse engineering of the Poly. By using it, we were able to investigate the Poly's networking code in order to to solve timing and hardware issues in our emulator. Our Poly emulator is now able to successfully emulate a Poly client attached to an emulated Proteus server.

The disassemblies that we created with Id will form the basis of a online library of information about the Poly's software and hardware. Because Id allows researchers to link and share information between their new disassemblies and our existing disassemblies of the operating system, BIOS, and other system utilities in our library, analysis of the Poly system can be achieved more rapidly and easily than with any other competing disassembly system.

The documents produced all contribute to a long-lasting store of knowledge about the Poly. The disassemblies are in a format that is readable even without using Id. They consist of plain-text, human-readable disassembly listings similar to what Id displays in its main pane, stored inside "zip" compressed archives. This ensures that the information discovered with Id will be accessible long into the future.

Our disassembler's Poly-specific knowledge is segmented from its dynamic document engine, so it is relatively easy to add support for more processors and architectures. This makes Id a very flexible tool for the disassembly and documentation of small systems.

References

[1] Various authors. *128K ZX Spectrum Technical Information*. World Of Spectrum, `http://www.worldofspectrum.org/faq/reference/128kreference.htm`, 2009. Accessed 17th September, 2009.

[2] Inc. Free Software Foundation. *GNU Binutils*. `http://www.gnu.org/software/binutils/`, 2008. Accessed 9 October, 2008.

[3] Ilfak Guilfanov. *IDA Pro Disassembler—multi-processor, windows hosted disassembler and debugger*. Hex-Rays, `http://www.hex-rays.com/idapro/`, 2009. Accessed 17th September, 2009.

[4] Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc., New York, NY, USA, 1977.

[5] R. N. Horspool and N. Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, Volume 23, Number 3, pages 223–229, 1980.

[6] Lance Lyon. *Commodore 128 Alive!* Commodore 128, `http://www.commodore128.org`, 2009. Accessed 17th September, 2009.

[7] The GNU Project. *GDB: The GNU project debugger*. `http://www.gnu.org/software/gdb/`, 2009. Accessed 15 September, 2009.

[8] T. Ritter and Boney J. The 6809. *Byte Magazine*, 1979.

[9] Steven Weyhrich. *Apple][History Chap 7*. Apple 2 History, `http://apple2history.org/history/ah07.html`, 2009. Accessed 17th September, 2009.