

PROTECTING SYSTEM AND SOFTWARE PATENT RIGHTS

Robert Krten, Edward Keyes, Vyacheslav Zavadsky
August 2008

INTRODUCTION

Almost every piece of modern technology, from cell phones and cameras to the automobile, now contains a microprocessor(s) and attendant software. Increasing amounts of functionality and value are now contained in the software portion of the system rather than in the hardware.

As the investments and value of this software portion increase, so too does the need to protect this IP against infringement. Central to any IP protection process is the ability to provide evidence of its use by third parties. Typically this requires obtaining samples of the potentially infringing appliance and analyzing the software inside.

In this whitepaper, we'll discuss and illustrate the different system and software analysis methodologies currently in use at TechInsights.

ANALYSIS METHODS

Approaches to system and software analysis can be grouped into three fairly distinct categories:

- Software Reverse Engineering (SRE)
- Software-Aided Functional Testing
- Architectural & Algorithmic Analysis

These methods can be used alone or in concert, depending on the nature of the problem.

SOFTWARE REVERSE ENGINEERING

Traditional SRE typically consists of extracting the complete memory contents of the appliance (the "binary image"), partitioning the binary image into different sections (code, text, multimedia, etc.), locating the relevant portion(s) of the code, followed by detailed analysis of those code sections.

There is a range of difficulty in obtaining the software in the first place – in some cases, it's simply available for download (or the memory can be physically removed and the data extracted readily), in other cases there may be a variety of protection mechanisms in place to prevent access. Hardware reverse engineering is usually effective in defeating these measures.

The detailed analysis of the software may consist of a simple disassembly of the binary file into low-level assembly language (sometimes referred to as "machine code"), or it may be necessary to perform a full "decompilation" (an attempt to get as close as possible to the original human-created "source code" written in a high level language). Assembly language is normally sufficient to determine a "who calls who" relationship hierarchy, whereas higher-level abstractions are usually required in order to clearly demonstrate algorithms.

Code analysis itself can be classified as "static" or "dynamic." In static code analysis, the code is simply disassembled and analyzed, whereas in dynamic analysis, the code is analyzed as it is being executed – running on real (or virtual) hardware. Dynamic analysis normally requires use of a debugger tool to step through the code, instruction by instruction, and correlate the behaviour of the hardware to the code, or simply to observe the path taken by the code.

Many different software tools are available to assist software analysis. In addition to the debugger mentioned above, these include:

- disassemblers (convert binary code into human-readable machine language instructions),
- decompilers (convert machine language instructions into higher-level language),

- simulators (allow the code to run in a virtual environment, allowing the human to change the simulated hardware characteristics dynamically),
- emulators (allow full control of the CPU hardware in a live, running environment)
- Software Development Kits (SDKs, provide architectural information about the chip, additional documentation, and often other software programs which are useful for analysis)

Generally speaking, disassemblers and debuggers (the lowest-level interfaces to code) are commonly available (usually for free) for just about every non-proprietary CPU on the planet. Decompilers are a relatively new technology, and are relatively scarce. Simulators are fairly common as well, but aren't always free / open source. Emulators are provided by the chip manufacturer and are usually quite expensive, and SDKs are also usually provided by the chip manufacture, with the cost varying.

By application of the above techniques, a working representation of the code can be created. Normally the results of this analysis cannot be directly used to map against a patent's claim language since patents by their nature claim higher level concepts. A further, higher level of analysis of the software is usually required. This is described in the next section.

ARCHITECTURAL & ALGORITHMIC ANALYSIS

Architectural & Algorithmic Analysis consists of looking a level above the actual software code and analyzing the operation or design of the software including any algorithms implemented in the software. Software reverse engineering (especially disassembly) gives only a relatively low-level view of the software's operation. Architectural and algorithmic analysis aims to look at how the aggregate software modules behave from a "system" level. Examples might include: an understanding of the fast forward operation in a DVD player or power train control in an automotive transmission controller. An excellent example of algorithmic analysis is given in the case studies section below, "Remaining Battery Life Estimation Algorithm."

SOFTWARE-AIDED FUNCTIONAL TESTING

In Software-Aided Functional Testing, specific system features of an appliance (frequently internal and often implemented in hardware) are analyzed for comparison to patent claims. Custom software routines are written and executed to exercise these system features and make them observable.

Examples of internal features might be an automatic power down of an internal block of circuitry after a set period of inactivity, operation of a microprocessor's internal cache, or changing of transmission/reception frequencies of a handset under certain conditions.

The software that's written for this purpose requires an intimate understanding of device operation. For example, after understanding how the device operates, code is created that will provoke a specific device behaviour if the patented feature exists in the device. The consequences of the behaviour can be observed in a variety of ways including monitoring variables such as execution time, the state of internal registers, or even device power consumption.

This analysis is normally conducted in tandem with hardware analysis and monitoring.

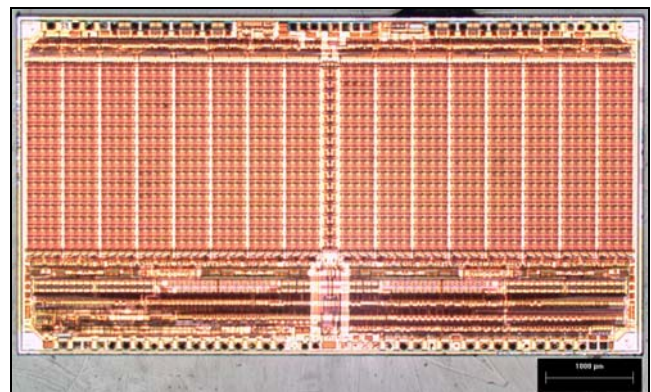
CASE STUDIES

The following case studies give examples of the different methodologies discussed above.

MICROCONTROLLER CONTROL OF A CAPPUCCINO MAKER

In this example we analyzed the control of steam generation in a cappuccino maker. The device under investigation used an embedded microcontroller to control the rate at which steam is released. The first challenge was extracting the code. The binary code on the microcontroller is normally inaccessible. This is an intentional security feature of that particular microcontroller: any attempt to write new code into its internal flash memory results in the circuitry of the microcontroller erasing the original contents of the flash.

Ordinarily, this would be an insurmountable problem – there's no way to get at the code in the



microcontroller in order to begin the SRE process. However, by combining forces with our hardware team, a solution was found and we were able to access the microcontroller's software.

A disassembler was not available for this particular chip, so we wrote one ourselves. Once we had the disassembled code in hand, we were able to analyze the algorithms used in the chip, and match them against the claim language of the patent.

REMAINING BATTERY LIFE ESTIMATION ALGORITHM

Today's cell phones contain many power saving features to extend battery life. An important function for the end user is an accurate estimation of the remaining battery life before a recharge is required. In this project, our task was to extract and reconstruct the battery estimation algorithm from a particular model of cell phone.



Smart phone

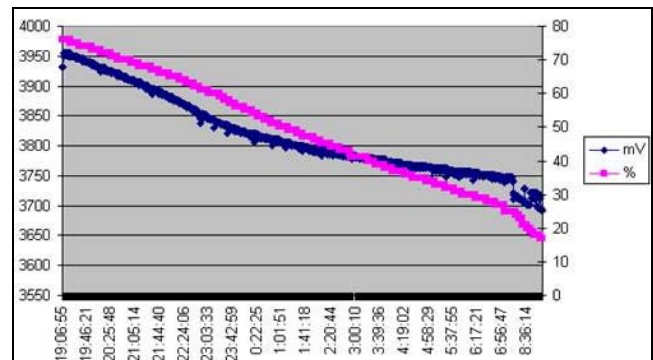
In a modern cell phone, it's not unusual for the entire binary image of the phone to be on the order of tens of megabytes – after all, it includes code for the DSPs, the application programs, the operating system, communication protocol stacks, sounds, images, and, of course, battery monitoring software. After extracting this huge binary image (which was in itself a challenge, as only part of the image was readily extractable), we analyzed it in order to find the portion that dealt with the battery. Finding the battery part of the code in tens of megabytes of binary image was accomplished in several stages. First of all, certain English ASCII strings present in the binary image led us to potential locations of the code – after all, anything that mentioned the word “battery” would

naturally be somehow related to battery monitoring code. Also, we had previously performed other RE work on the software, and determined where a number of the key data structures were located. Some of these data structures pertained to the battery code, so it was a matter of tracing the references to those data structures to find the code that dealt with the battery.

Comparable algorithms use a reasonably simple mathematical formula, so we anticipated finding only a few kB of battery code. However, our analysis showed that the amount of code dealing with the battery on the cell phone was around 70k bytes. After disassembling the code, we discovered part of the algorithm, but also found that part of it relied on data provided by an inaccessible, proprietary DSP. Analysis of the DSP hardware (required in order to fully understand the algorithm) was possible, but prohibitively expensive.

We solved the problem by complementing the software reverse engineering with architectural & algorithmic analysis. We wrote a program to monitor estimates of remaining battery life and downloaded the program to the phone. This program periodically read the memory area that contains the phone's estimate of remaining battery life (known to us through the traditional SRE done in the first phase of the project) and then saved it to a file on the phone.

We simultaneously monitored the battery's actual charge state using a sampling voltmeter. Knowing the actual and estimated battery life values allowed us to reverse engineer the estimation algorithm and compare its behaviour to the predictions of the patented algorithm. This project was interesting in that the high-level functional analysis turned out to be simpler and faster than a full SRE approach (assuming we had access to all the code in the first place). Trying to deduce the original mathematical

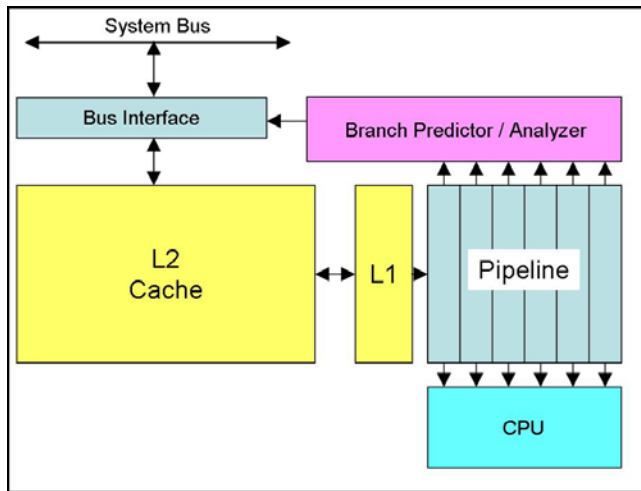


Battery Discharge over Time

formula via SRE would be time consuming – however, simply reading the data generated by the existing software, and comparing the values to those from the customer’s patented formula was much less work.

CACHE BEHAVIOUR IN A MICROPROCESSOR

Today’s leading-edge microprocessors, contain multiple processor cores. They use complex caching algorithms to optimize the operation of the processor’s multi-stage pipelines and cores. It’s all about execution speed – keeping the instruction pipelines full of executable instructions, and predicting the location of the next required code or data to keep the cache one step ahead of the pipeline. Numerous patents are in place that embody different strategies for accomplishing this task.



Typical processor architecture showing cache and branch prediction

One of our recent projects in this area was to investigate patented cache algorithms involving a certain type of “speculative prefetching” (guessing what the next required code or data will be and pre-loading it into the cache). The patent called for a prefetch of code and data to take place after a conditional branch instruction – effectively, the processor would “look ahead” and start getting data even when it wasn’t yet certain that it would be required. If it turned out the data was in fact required, then the entire operation would have been performed in less time, because the data was already present in the cache.

Doing this through pure hardware reverse engineering would have been time consuming and expensive – the internal architecture of the cache

manager would have to have been reverse engineered and simulated. Instead, we approached the problem through software aided functional testing. The main challenge was in determining how to prove whether or not the speculative prefetch was happening.

When a prefetch occurs, it acts just like any other memory access. That is, the processor causes a read cycle to occur on the external bus, and data to be read from the external memory into the processor. To speed up the processor’s execution, the cache memory is updated with this data at the same time so that if the data is required again in the future, it can be fetched from the (much faster) cache memory rather than external memory. The basis of our approach (to determine if the prefetch occurred) was to look for “side effects” of the prefetch – in this case, the memory contents of the cache after execution of the test routine.

The test routine we developed was written to first initialize the processor’s cache to a known state by performing a series of reads from a dummy memory region that would, by design, not contain any required data. The read size was an order of magnitude bigger than the cache, thus ensuring that the cache would be filled with data from the dummy memory area. The next part of the test routine contained a series of hand-crafted assembly language instructions containing two different paths. The code was designed such that, to the processor, both paths looked equally likely to be executed. However, we designed the code such that only one path would ever actually be executed. The second (non-executing) path was selected such that it would touch another area in memory, which we dubbed the “evidence” area.

The crux of the test was the following: if the processor performed the speculative prefetch, then, at the conclusion of the test, locations from the evidence area would be present in cache (because the processor would have “speculatively” read them into the cache).

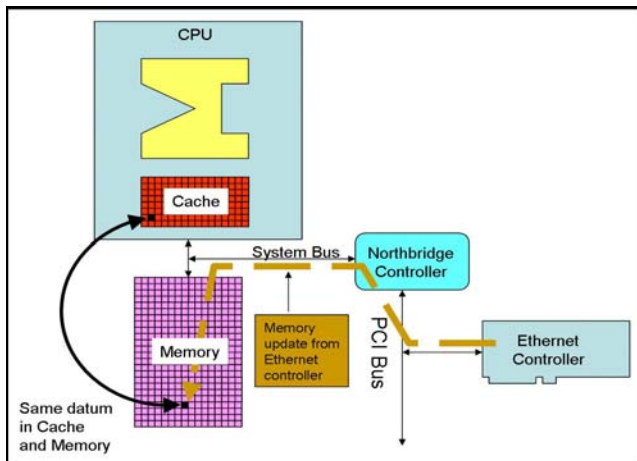
The processor doesn't provide any means to directly read the contents of the cache (to verify that they held portions of the "evidence" area). We could, however, indirectly verify the contents of the cache by reading data from the evidence area (at the conclusion of the test routine) and measuring the read speed. A "fast" read would indicate that the data must already have been in the cache and that the processor had therefore performed a speculative prefetch. A slow read would indicate that the data was not present in the cache and hence, no speculative prefetch. Values for "fast" and "slow" read operations were obtained beforehand, by timing the execution of similar code.

Thus, through clever test routines and by monitoring read speeds, we were able to test for the presence of a speculative prefetch along the lines of our customer's patent.

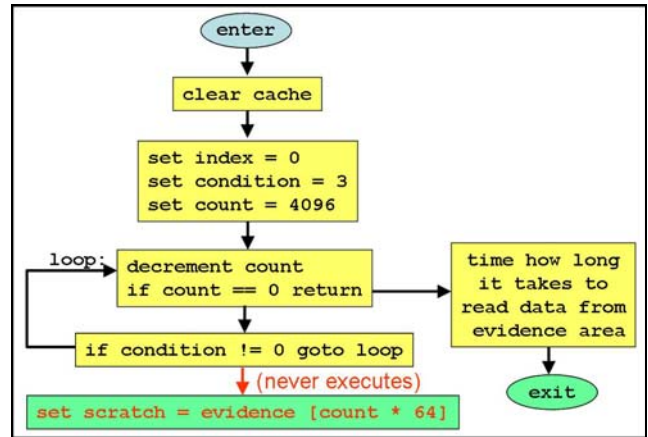
NORTHBRIDGE CONTROLLER BEHAVIOUR

Modern processors contain cache memory to improve execution speed. Frequently-used code and data are stored and retrieved from the fast cache memory rather than the slower main memory. Since the price of memory is directly proportional to its speed, a typical PC can afford lots of "slow" (main, external) memory, typically on the order of gigabytes, but may only be able to afford a few megabytes of very fast (cache, internal to the processor) memory.

One difficulty with cache memory, however, is when an external peripheral changes the contents of external memory (for example, as a result of data arriving on a hardware port). There needs to be a mechanism that allows the CPU to update its internal cache memory with the new contents of the external



Ethernet controller updates main memory via DMA operation



Main algorithm for exercising speculative prefetch side-effects

memory. This is called "maintaining cache coherency."

Cache snooping is an operation that ensures cache coherency between the main memory and the CPU's cache. Cache coherency operations ensure that the cache is updated whenever data is written directly to the main memory by a peripheral device. Similarly, the data in main memory is updated whenever data is written to the CPU's cache. This ensures the consistency of the data in both the cache and main memory.

The objective of this project was to look for the presence of a cache "snooping" signal on a PC motherboard. When the PCI bus master device writes data to an address location mapped to the cacheable part of main processor memory, a "snooping" signal should be generated on the processor FSB if snooping is used. The snooping signal is detectable with a logic state analyzer or oscilloscope.

The test setup used in this investigation was a basic PC running Linux. An Ethernet card was selected as the target device due to the simplicity of generating data for it, and the fact that the device driver for it was relatively easy to modify. The hardware part of the test consisted of connecting a logic analyzer to pins of interest on the CPU and the PCI bus (especially the "snoop active" line), and observing the results.

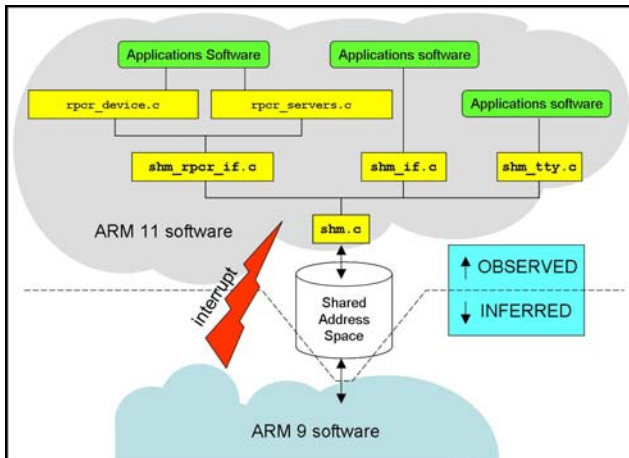
External memory can be tagged as "cacheable" or "non-cacheable." When memory is tagged "non-cacheable," it means that the CPU will always read directly from the memory itself, never from a cached value.

The software part of this project consisted of modifying the Linux Ethernet driver. Two versions of the driver were created, both using a target memory area where data from the Ethernet card would be automatically transferred as soon as it arrived. One version of the software declared the memory area as “non-cacheable,” and the other version declared that same memory as “cacheable.” When run with the non-cacheable attribute, we expected that no snoop signal would be generated, because the CPU wouldn’t be updating its cached version of the data. When run with the second version, which declared the memory area as cacheable, a snoop signal would be expected if snooping was implemented on the targeted motherboard.

As a result of the testing we were able to successfully render an opinion about the use of snooping for our client.

ANALYSIS OF LINUX SOURCE CODE FOR INTER PROCESSOR COMMUNICATIONS

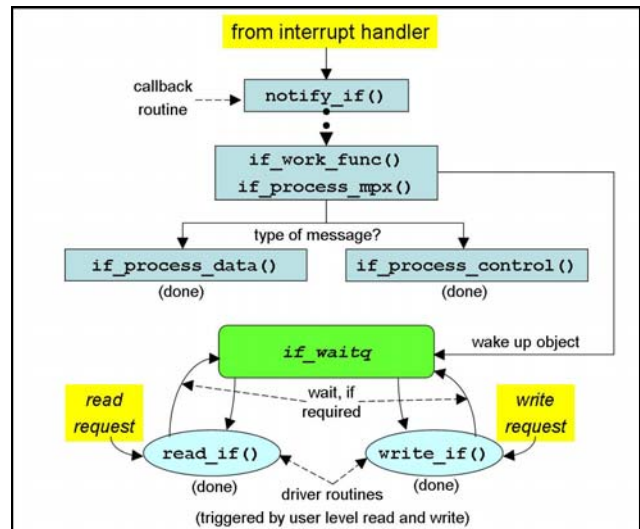
In our final case study, we were asked to analyze a large open source project to determine if a contributing party was using patented technology related to inter processor communications. At a high level, inter processor communications concerns the communication between two processes running on different processors – the processors may be on the same die (as in a multi-core device), on the same board, or even on different systems entirely.



High level architecture of IPC system

We obtained just over 8 million lines of source code in just under 21 thousand files consisting of a Linux kernel distribution for a cell phone. Within a short period of time, we were able to identify an area of interest comprising around 4,500 lines of code in a dozen or so modules. This area dealt with a specific device driver which provided an RPC stack to talk via a shared-memory based “window” to other processors on the same device. RPC stands for “Remote Procedure Call,” and is the ability for one process to call functions from within another process, whether that second process is on the same processor, a different processor core on the same chip, or halfway around the world over an Internet connection. Auxiliary code provided network and console functionality.

In this project, the work was to analyze the overall architecture of the code – the end-to-end message scheme and notification scheme. A major challenge was that the other end of the communications pipeline was connected to one or more proprietary DSPs and another processor (well documented, but for which we didn’t have any code). We started by mapping out the shared memory areas, and seeing how they were used. A large, one megabyte data area was discovered which turned out to have multiple channels of data, organized along ring buffers with a head and tail for each channel. Next, we discovered how the main processor communicated with the remote processors – there was a hardware line that was set in order to tell the remotes that data was available in the shared memory region. And, in the reverse direction, interrupt lines were raised by the remotes in order to wake up the main processor.



Interrupt handling of dispatch

At a higher level, we then examined several clients of the RPC stack (network and console drivers) to understand how they used the provided functionality. The end result of this was a report detailing the overall software architecture of the RPC stack and

how it interacted with the remote processors, and how the stack's clients were able to achieve end-to-end communications. Some facets of this RE work were also used for the Remaining Battery Life Estimation Algorithm project, above.

ABOUT UBM TECHINSIGHTS

UBM TechInsights is the preeminent firm for the provision of sophisticated information and advice to technology companies. Our foundation of technical expertise and analysis tools, combined with years of experience on all aspects of the IP / Technology Lifecycle allow us to deliver the highest ROI to the client. Our leadership position is sustained by our commitment to the upmost integrity, and the ongoing investment and acquisition of knowledge and capabilities necessary to meet our clients' evolving needs. If you need assistance in managing your most complex technology or IP challenges, turn to UBM TechInsights for assured results. Visit www.ubmtechinsights.com.