Assessment Of Paper:

VPFS: Building a Virtual Private File System with a Small Trusted Computing Base

By Carsten Weinhold and Hermann Hartig

Student Number: ????????

1. INTRODUCTION

With the motivation to provide confidentiality, integrity and recoverability to security critical files, such as those used by banking programs or email clients, this paper describes a process whereby a multi-level secure system is implemented to provide secure isolation between a small trusted computing base running trusted applications and the VPFS server, and a non-secure virtualized operating system interfacing with an untrusted permanent storage device. This approach mainly operates under the belief that a small trusted computing base will have far fewer bugs, and therefore less vulnerable than than alternative schemes that form part of the legacy OS, such as Microsoft Vista's Bit Locker technology.

2. APPROACH

The approach documented in this paper aims to minimize the size of the trusted computing base (TCB) by only running small, trusted applications inside the TCB, and executing the untrusted commodity OS on top of a microkernel, such as L4. This approach makes the assumption that a smaller TCB is likely to have less bugs and provides a much smaller area from which to attack, making it inherently safer, which is a reasonable assumption to make. It is mentioned that each application is only trusted for its individual purpose, but it is unclear how this system would operate if more than one application is required to be in the TCB, as described in section 4.

A threat model is also presented, which provides a nice assessment of possible sources of attacks, leading to an assumption that the trusted computing hardware is tamper resistant. While there is at least one known attack that can potentially reveal the contents of the trusted computing module, as will be described in section 4, given the current status of security in industry, I believe this is a valid assumption to make. Congruously, this method requires the hardware to actually contain a trusted computing module. Again, I believe this is a fair assumption to make, as most hardware manufactures started to include them in chips since around 2004, and there are many examples of other file system encryption methods that use the TCM, such as the Bit Locker Drive encryption used in Windows Vista.

The major approach taken by the authors is splitting the VPFS code into a trusted VPFS server, and an untrusted VPFS proxy, that runs in the legacy OS and interfaces to the untrusted storage, thereby removing file system code from the TCB. The VPS server interfaces with the trusted application, and encrypts the data, before sharing this data with the proxy for writing to the disk. Although it is not mentioned, presumably care was taken to make sure the untrusted VPFS proxy never has access to the unencrypted data, such as removing access to the

area of memory during encryption, or placing the unencrypted data in another memory location to the one the VPFS Server and Proxy share.

Individual files are wrapped in file containers, which are encrypted by a secure, blocked-based cipher using the AES algorithm and a collision resistant hash function. This system ensures confidentiality as long as the master encryption key is protected in the PCM. Integrity of the data inside the file container is delivered using an embedded tree based hash. This approach uses strong cryptographic protocols to ensure a high level of security – it is clear that the goals of confidentiality and integrity of file data has been achieved. However, as pleasingly acknowledged in the report, this approach dos have some downsides, being:

- The tree-based block structure in a file container introduces a parent-child relationship, which causes the entire file container to be invalidated if one block is compromised, such as if the system crashes before all nodes in the container are written to disk.
- Wrapping confidential files in file containers does not hide approximate file sizes (known to the nearest block size), the number of files or the time stamp from untrusted users. The authors here assume that this small compromise to confidentiality is ok for the majority of cases. Perhaps the authors could consider a modification where files are wrapped in fixed size file containers, with several files potentially occupying a single container, or a single file spreading across multiple containers. While such an implementation would raise the complexity of the code in the TCB, it would provide greater confidentiality if needed.

The hash sums of each file container are cleverly stored in a master hash file, with only the master hash sum being required to be stored in the TCM. It is assumed that the performance hit of needing to use another file for authentication won't be significant as the master hash key file can be stored in the VPFS server's cache, however, as discussed in sections 3 and 4, I do not feel that scalability concerns are fully examined in the paper.

Finally, the authors present a detailed discussion on two different approaches on implementing file naming and directory structure in the secure file system:

- Untrusted naming: This approach minimizes code needed in the TCB, but only supports the concept of a hierarchal namespace, whereby the path to a file is encrypted as part of its name. Despite this drawback, the authors go to a great deal of trouble to show that this approach meets a set of stringent integrity security properties.
- Trusted directories: Unsorted filenames are stored in per-directory file containers, which can be looked up using a linear search. It is easy to see that this system also meets the intended security properties. More complex lookup strategies were not implemented in order to keep the TCB as small as possible, and the performance losses are partially offset by using a lookup cache. I do question though whether a relatively small amount of extra code to provide a faster lookup would be worth the slight worsening in code complexity, particularly in cases with complicated file system structures. To a certain extent, this issue links back to my previously expressed concerns about scalability.

The approach also included a few optimizations to maximize performance, such as a buffer cache, and the previously mentioned lookup cache. All these systems certainly have performance benefits, but come at the cost of more complex code. While I agree with the author's position that the performance gains are worth the small gain in TCB size, given shortcuts in other areas (such as the linear search in trusted directories) were taken to

keep the TCB to a minimum, it would have been nice to see the performance gains of each potential optimizations so the best could be chosen, instead of the current ad-hoc inclusion approach.

3. RESULTS

Firstly, the authors calculated the lines of code their solution and its cyclomatic complexity, which succeeds in showing that they have achieved their goals of minimizing the TCB. It should be noted however that these figures do not include the microkernel part of the TCB. Even so, this solution has orders of magnitude less code than solutions which are based directly in Linux or Windows.

For testing, L4Linux was installed on top of an implementation of the L4 microkernel called Fiasco. The VPFS proxy is run in L4Linux as a virtualization-aware Linux program. Several other programs providing similar level of security, such as dm_crypt and EncFS, as well as an unmodified file system were all set up to run a set of benchmarks on the same hardware. Furthermore, the other applications tested were run both of native Linux and L4Linux to show the performance effect of the microkernel. This is a very good and thorough experimental procedure, and the author's clearly went to a lot of trouble to demonstrate the relative performance of their system. The underlying file-system used is ReiserFs 3.6. While this is a reasonably popular file system, I would like to have seen results from a more widely used file system, such as ext2 or 3. One point of interest is that for the VPFS machine, they split the memory into 64 MB for the VPFS Server, and 192 MB for L4Linux. As they mention one of the goals is to be able to run this on mobile hardware, where memory resources are limited, it would have been nice to see performance effects of reducing the VPFS amount, as 64 MB is a surprisingly large chunk of the memory (probably designed to ensure all the necessary files for the tests can be kept in the VFS Server's buffer).

Benchmarks were performed to read and write a very large file, many small files, and a string search across files of various sizes. However, the second benchmark, still only used a few hundred files. Due to the previously mentioned performance concerns of the directory implementation and the buffer size, it would have been interesting to see the effect of many thousand files on performance to give us some idea about the scalability of the system.

While largely impressed with the quality, detail and effort present in the author's evaluation of their technique, one area completely missing was security. While reasonable performance is clearly vital for a security method (if it is too slow, it will not be used), and minimizing complexity in the TCB is a great way to provide the *potential* for a much more secure system, I feel that some analysis of exactly how secure the VPFS system is relative to its counterparts. It wouldn't hurt if some penetration tests were performed to show how hard the system is to break, or to show some examples of attacks that can get through the other methods, but are stopped or detected by the VPFS system. The argument that a smaller TCB is less likely to have bugs isn't strong enough to show that their method is actually more secure than other methods.

4. ANALYSIS OF ISSUES

I feel that the paper presented a novel unique method to achieve secure and trusted storage on untrusted hardware that can run untrusted software, and went to a great deal of effort to explain their method and give

some results. To this extent, they were largely successful, but I feel that there are a few issues that need to be addressed, that have not been mentioned in the previous sections.

Firstly, the major goals of VPFS is to achieve confidentiality, integrity and recoverability. Indeed one of the concluding statements is "VPFS leverages trusted computing technology and is able to ensure confidentiality and integrity – including freshness – of all file system data and metadata." While I believe the integrity goal has been fully realized (though as mentioned previously, no tests were attempted to show this), the confidentiality goal has only been largely realized – certain aspects of the metadata, such as the size, time, and number of files, are not confidential, making the concluding statement somewhat misleading. Admittedly, in the vast majority of use cases, this degree of information leakage is alright, but it could potentially lead to some problems. For example, the master hash file would be fairly easy to identify. Also, if using a banking application that only runs its secure services in the TCB, files related to banking would be easy to spot.

Perhaps more important is the goal of recoverability. It is mentioned several times that due to space constraints, recoverability is not included. However, as one of the three major goals for the project, I find the absence of discussion on recoverability, albeit a brief mention on using external storage to be worrying. Additionally, their solution being structured around using tree hashes, and a single node to store all the file hashes make the private system particularly vulnerable to either unclean shutdowns where not all the data is written to the disk, or corruption/data loss of the disk. It is briefly mentioned that data can be backed up off site, but there remain questions such as how often it happens, how it works, how it guarantees security, etc. Possibly one of the biggest problems with the backup idea is the use of the TCM – the decryption keys rely on the hardware remaining constant. In the event of a system failure, or theft, it is impossible to decrypt the data, placing serious concerns on the recoverability goal of the system.

Furthermore, while the use of a TCM is considered acceptable for high security, there are possible attacks which could render the security void, such as the fact that the keys would be stored in memory during booting. It was shown earlier this year that, due to DRAM retaining memory for several minutes after being powered off, the machine could be turned off during boot-up and booted using a USB device, and a memory dump taken. Potentially more troublesome, however, there remains a question over side attacks – presumably some sort of authentication, such as a password, would need to be provided for a user to start running and accessing an application in the TCB. Because this is occurring from the untrusted side, a simple keylogger could steal the password and gain malicious access to the trusted side. Additionally, there also remains a question as to how applications actually can get in the TCB, and if these applications can be trusted.

Finally, I also remain unconvinced about how multiple trusted applications would work. It is mentioned in the paper that each application is only trusted for their purpose, and later that applications interact with their respective VPFS server, but no detail is provided about how this works. Presumably, each application runs its own VPFS server to prevent applications from possibly interfering with each other. However, as the security of the entire file system relies ultimately on information stored in the TCM, and given the TCM's limited resources, I would imagine each VPFS would have to use the same information, giving each trusted application would therefore have access to the entire file system.

5. CONCLUSION

I feel that this paper addressed the problem of providing secure data storage using only a minimal TCB and untrusted hardware. The authors provided a detailed and compelling explanation of their method, and to a large extent I believe they have been successful in producing a usable and highly secure VPFS. However, in addition to a few issues regarding the method, such as the recoverability, the lack of tests to demonstrate the security properties of the system is troublesome.

Assessment Of Paper:

Documenting and Automating Collateral Evolutions in Linux Device Drivers

By Yoann Padioleau, Julia Lawall, Rene Rydhof Hansen and Gilles Muller

Student Number: ????????

1. INTRODUCTION

This paper seeks to provide a solution to shorten the hugely time consuming process of updating Linux device driver files when a change is made that affects an API. This process of collaterally evolving potentially 1000s of files is currently performed by hand, using only basic tools such as grep and sed. The authors propose using an automated transformation tool called Coccinelle, which presents a language based on normal patch syntax to create a *semantic patch*, which is then automatically applied across the Linux source tree.

2. APPROACH

The authors begin their approach by enumerating several challenges their tool must meet: ease of use; preservation of coding style; genericity; and efficiency. To do so, they exploit and extend the current heavy use of patch files in the Linux community. An assumption is also made that it is tractable in most cases to directly parse code containing preprocessor directives through the Coccinelle patching engine. This is clearly the best approach for simplicity and understandability of semantic patch files, however a more in depth discussion on cases where this assumption fails, and an evaluation of how often this happens would be appreciated.

The overall approach of Coccinelle consists of a library developer, after making a change to an API beginning a modify – test – examine – repeat system whereby they make a modification to the semantic patch, apply it to the affected drivers, examine the cases where the patch applied incompletely to affected files, and refine the patch. This continues until all affected driver files are correctly patched. It is postulated that at this stage maintainers of drivers outside the source tree and/or motivated users can then apply the semantic patch to driver files outside the Linux source tree.

Coccinelle defines a language based on patch file syntax called smPL, whose core principles are demonstrated effectively by using an example driven approach. A further detailed set of examples is provided in an appendix. In addition to syntax designed to allow a series of rules and line inclusions/deletions, the language also cleverly detects isomorphisms, and makes the change required while keeping the style of the isomorphism used before. An extension to the rules concept allows for a generic global search and replace to be implemented.

Control flow variations are dealt with in a much more complex manner, making use of temporal logic and control flow graphs to make the required changes. It is noted in the results experiments section of the paper that in more complex cases, this process has the potential to get stuck and time out. Finally, pre processor directives are managed through using a series of heuristics to predict how a change should occur. It is

mentioned that these heuristics have been shown to work 99% of the time, with the remaining problems largely due to complex code patterns, which is a very impressive result.

This approach presents an extremely well thought-out and considered approach to automatic applying of patches. The language used is simple to learn and read, yet is powerful in its capacity to deal with complex scenarios such as pre-processor directives and isomorphisms. Pleasingly, the authors also identify areas in which their process does not work so well – mainly when it has to deal with data-flow relationships and inter-procedural control flow relationships, and this functionality is being added. It should also be noted that even if the semantic patch is unable to provide a full, working patch, it will often still detect a partial match, allowing the user to be notified so a patch can be applied manually.

3. RESULTS

The general approach used for testing was to find a large number (62) of instances where collateral evolutions took place in Linux source code, ranging from simple to complex, and applying a semantic patch to the source tree before the evolution occurred. Although testing this software is a potentially difficult and time-consuming method, this approach is a great way to build up a series of test cases in a relatively short amount of time.

However, I am not sure why the authors did not attempt to follow the procedure they detailed in their approach more closely. Collateral evolutions were identified by using a tool called *patchparse*, to detect commonly occurring changes. From these changes a semantic patch was built up. It would be interesting to see the affect building up a patch based only on an API change would have, as is done in real life, as there is potentially less information available. Furthermore, the patch is applied only to the files identified by patchparse, as opposed to the normal use method of applying it over the entire source, which could cause additional files matched and changed (that may or may not be supposed to be changed) and the time taken to apply the patch – taking time measurements over a subset of files is quite meaningless to the overall efficiency of the method, yet no complete readings were reported.

Furthermore, the times presented in the table are the average running it takes Coccinelle to apply the semantic patch to a single file, which is not a good measure to demonstrate the overall efficiency of the system. This is compounded by the duration column in the manual evolution section of figure 7, as these figures are not an accurate comparison of the relative time taken to apply the patch, as sometimes less important changes are not implement straight away. I also question the use of comparing lines of code in the patch against lines of code in the semantic patch as an evaluation technique, as well as being interested as to how the authors worked out the errors or missed collateral evolution sites in the manual evolution process.

Perhaps a better comparison of the efficiency of the alternative methods would be to time a developer updating collateral evolutions across the Linux source tree against another developer using Coccinelle to perform the same changes, as this, despite potentially introducing inaccuracies, would provide a meaningful comparison of the overall speed of the different methods.

Finally, the results don't seem to specify how often partial matches occur, only how often misses occur. This is important as a partial match directs the programmer to files to examine, which is better than files being completely missed. Additionally, there was no mention of analysis made to ensure the integrity of any changes

made – it would be interesting to test if the patches ever incorrectly changed a file. Despite these small concerns, I feel that a thorough set of results was presented across a wide range of possible cases, which adequately reported the percentage success rate of the proposed method.

4. ANALYSIS

This paper identifies a real problem operating system developers have when they change the API, potentially encouraging more worthwhile changes to the API to be made due to easier updates of affect code. Additionally, there does not seem to be a tool that is able to provide automatic program transformation to low level code such as C. The method proposed is well thought out to fit in with current Linux programming methodologies, cleverly ensures style is maintained through changes, and is able to deal with changes to complex pre-processor directives in the vast majority of cases.

However, there are two major potential issues with the method that could prevent its immediate acceptance among the Linux community:

- The process of developing the patch could potentially be time consuming, and it relies on the developer understanding the Coccinelle syntax, and taking the time to test and retest the semantic patch until it correctly fixes the vast majority of driver files. For complex changes, the semantic patch file starts to become very complex, and great care would need to be taken to make sure the patch never incorrectly changes files. As no direct timing comparisons with the old method of collateral evolution using grep and sed were made, the complex development of the patch file, and the complex checks to make sure the changes are correct and have been applied where required, might actually make the manual system faster.
- It is stated in the abstract that changes were able to be correctly applied on average 93% of the time. However, when making modifications to potentially 100s of files, this represents a fairly significant miss rate – certainly to have confidence that all the required changes have been made, you would have to search through all the files not changed to find files that were missed by the automated system. As mentioned above, complex changes might also necessitate going through all files that were changed to make sure they have been correctly modified. In doing both of these steps, the proposed method may actually take longer than the original manual method.

5. CONCLUSION

I feel that this paper makes significant positive advances in the field of automated collateral evolution by presenting a unique, simple and very well thought language to allow automated propagation of required changes. While reasonably high miss rates, possibly large due to acknowledged weaknesses in dealing with data-flow and intra-procedural control flow relationships, may limit the use of the Coccinelle method in its current state, it has a strong potential to be able to be quickly used to provide fast and accurate collateral evolution in the future.