Matryoshka: Strengthening Software Protection via Nested Virtual Machines

Sudeep Ghosh
Microsoft Corporation,
One Microsoft Way, Redmond, WA 98052
Email: sugho@microsoft.com

Jason D. Hiser and Jack W. Davidson
Department of Computer Science
University of Virginia
Charlottesville, VA 22904
Email: {hiser, jwd}@virginia.edu

***Abstract*—The use of virtual machine technology has become a popular approach for defending software applications from attacks by adversaries that wish to compromise the integrity and confidentiality of an application. In addition to providing some inherent obfuscation of the execution of the software application, the use of virtual machine technology can make both static and dynamic analysis more difficult for the adversary. However, a major point of concern is the protection of the virtual machine itself. The major weakness is that the virtual machine presents a inviting target for the adversary. If an adversary can render the virtual machine ineffective, they can focus their energy and attention on the software application. One possible approach is to protect the virtual machine by composing or nesting virtualization layers to impart virtual machine protection techniques to the inner virtual machines "closest" to the software application. This paper explores the concept and feasibility of nested virtualization for software protection using a high-performance software dynamic translation system. Using two metrics for measuring the strength of protection, the preliminary results show that nesting virtual machines can strengthen protection of the software application. While the nesting of virtual machines does increase run-time overhead, initial results indicate that with careful application of the technique, run-time overhead could be reduced to reasonable levels.**

I. Introduction

Virtualization has become a popular technique for protecting software applications from compromise by malicious adversaries who wish to either tamper with an application or reverse engineer an application to steal intellectual property or recover other assets that exist in the application. Examples of tools that employ some form virtualization to protect software include Themida [1], VMProtect [2], Strata [3], and CodeVirtualizer [4]. While virtualization does provide protection, a point of concern is that the virtual machine itself may not be well protected. An adversary that can reverse engineer or tamper with the protective process-level virtual machine (PVM) may then be able to directly attack and compromise the software application. [1]

One possible solution to this problem involves nesting layers of virtualization to impart protection to the PVM. Figure 1 provides a conceptual illustration of an application protected by nested PVMs (N-PVMs). Here, PVM_1 protects PVM_2, which protects the application. To attack the application, the adversary would have to break through a protected PVM layer to reach the application. As should be obvious, the level of nesting can be made arbitrarily deep, although run-time overhead issues must be considered.

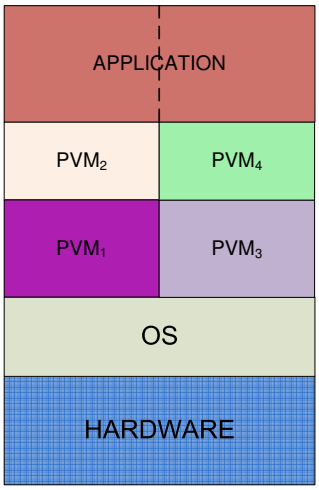

Fig. 1. A conceptual overview of a N-PVM-protected application package.

Figure 1 also illustrates the ability to apply different N-PVMs to different sections of code. PVM_1 and PVM_2 protect one section of the application, while PVM_3 and PVM_4 protect another section of the application.

Nesting of PVMs and partitioning of the application provides flexibility to the defender. Key sections of code can be protected by N-PVMs, while other sections of code may execute without protection (e.g., code that does not contain critical assets).

The contributions of this paper are:

- Introduction of a novel protection technique, called *nested process-level virtual machines*, or N-PVMs. In this scheme, an application is partitioned, and a nested set of virtual machines are assigned to protect the application partition and themselves.
- The presentation of a case study to illustrate some of the protection benefits of this technique and to gauge the feasibility of N-PVMs.

[1] This paper presents material from the Ph.D. thesis of Sudeep Ghosh, *Software Protection via Composable Process-level Virtual Machines* published in December 2013.

978-1-4673-7094-3 2015

U.S. Government Work Not Protected by U.S. Copyright
DOI 10.1109/SPRO.2015.11

- The design of an optimization technique to reduce the performance overhead of software dynamic translation associated with multiple layers of virtualization translation. This optimization is shown to reduce run-time overhead significantly.

This paper is organized as follows. Section II briefly discusses application partitioning. The methodology for creating nested PVM-protected applications is presented in Section III. Section IV presents measures of the protection properties of N-PVMs, and it also presents preliminary measurements of the run-time overhead of N-PVMs. This section also discusses an optimization that significantly reduces the run-time overhead associated with N-PVMs. Related work is discussed in Section V, and Section VI provides a conclusion and directions for further research.

II. Partitioning Applications

The first step to create a N-PVM application is to partition the application, and to assign each partition to be protected an N-PVM, (or allow the partition to run natively without protection). At runtime, each protected partition will run under mediation of its assigned N-PVM.

Determining the appropriate partitions of an application depends on numerous factors including what information should be protected, which portions of the application contain that information, overhead concerns, and the level of stealthiness desired. Previous research has investigated the appropriate partitioning of an application for security purposes [5]–[8]. In this paper, we assume that some mechanism, including manual partitioning by an expert software engineer, has determined which sections of an application should be protected and at what level.

III. Nested Process-level Virtual Machines

The N-PVM infrastructure presented in this paper is based on a high-performance software dynamic translation system called Strata [9]. Strata operates directly on executables. As an application is executed, Strata can apply various transforms to protect the application code as well as itself [3], [10]. In addition to Strata, we utilize the Diablo link-time toolchain [11].

Each N-PVM instance for a partition is created from the Strata library. To create the multiple N-PVM instances, we employed the `objcopy` utility from the `binutils` suite of tools. `objcopy` copies the contents of one object file (including library and executable files) to another, while providing options to modify the output in several ways. One of the options involves modifying the names of all the global symbols in the output file by adding a prefix. This option can be used to ensure that the different PVM instances do not share any global symbol names, thus removing any name conflicts. Utilizing `objcopy` provides a simple technique to create several PVM instances, without performing extensive source code changes. It is important to note that creating N-PVM instances with different protection properties, some amount of code modification may be required. Discussion of the creation different PVM instances is beyond the scope of this paper. For simplicity and as a worst-case evaluation of protection strength, each N-PVM instance possesses identical protection properties. Furthermore, a single N-PVM is applied to the entire application, which approximates worst-case, runtime performance.

Strata has three functions that are of relevance. The `strata_init` function initializes the Strata library, the `strata_start` function virtualizes the application, and the `strata_exit` function deallocates any resources that Strata might be using, and relinquishes execution control back to the controlling execution context, which may be the native hardware.

Once all the object files and PVM instances are created, they are provided as input to Diablo. Diablo processes the inputs and creates the CFG for the application. At this point, Diablo is used to insert calls to `start_init`, `strata_start` and `strata_exit` for each PVM instance at appropriate locations in the CFG corresponding to each partition. If a PVM instance is itself to be protected by a PVM (i.e., nested execution), Diablo inserts the appropriate calls to the PVM accordingly.

Next, Diablo generates machine code for this modified CFG and produces an executable file. Various static protections can also be applied to the executable at this stage. The resulting output file is a N-PVM-protected application binary. At run time, when a call is made to `strata_start`, the corresponding PVM instance starts mediating the application. When a call is made to `strata_exit`, the corresponding PVM instance releases its resources, and yields control.

These tools and techniques provide a flexible platform for the software defender to craft and apply different configurations of N-PVMs. For example, the defender may choose to partition the application such that only one function runs under control of a N-PVM, or that multiple partitions are protected with different N-PVMs, each one employing different protections.

To obtain insight into the protection offered by N-PVMs, we performed a case study that consists of an unprotected application, P_{APP}, and two PVM instances, Strata_1, and Strata_2. Strata_1 translates Strata_2 and the translated Strata_2 translates the application.

Figure 2 illustrates the creation, and the run time of an application protected by the N-PVM. As before, during software preparation, Diablo generates the CFG of the application. In this case, all the blocks comprising `main` are encapsulated by the start and exit functions of Strata_2. Consequently, Strata_2 is encapsulated by Strata_1, ensuring that Strata_2 executes under the control of Strata_1.

At run time, Strata_1 assumes control. It proceeds to translate code from Strata_2 to its software cache, SC_1. Then, control is transferred to the translated code in SC_1, which in turn, starts translating the application's code to Strata_2's software cache (SC_2). As control is about to be transferred to SC_1, Strata_1 captures control, and starts translating code from SC_2, to its own cache, SC_1. Thus, all the instructions that execute natively belong to Strata_1 and reside in its software cache, SC_1. This

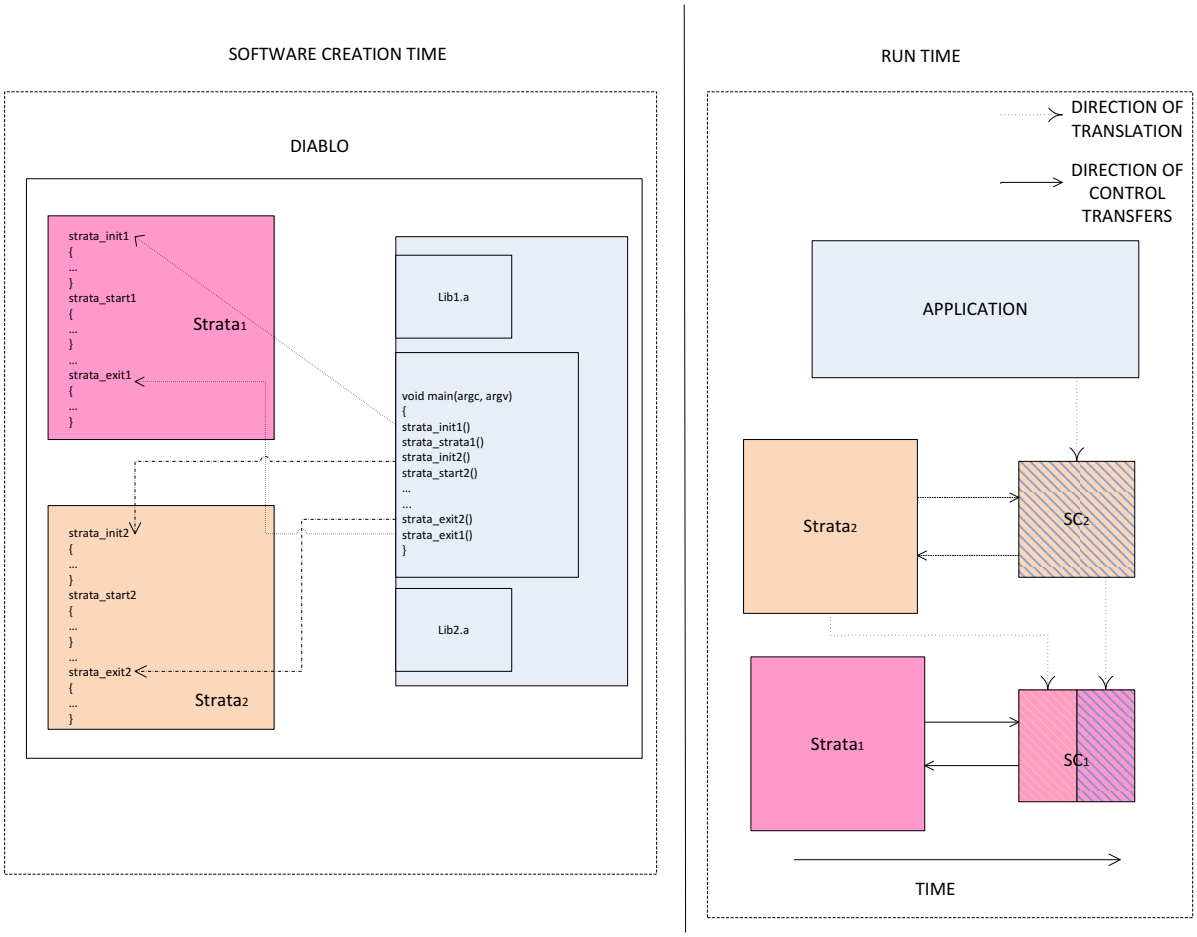

Fig. 2. A high-level overview of the nested N-PVM configuration. During software creation, Diablo synthesizes the CFG of the application and inserts the entry and exit functions of the PVM instances, Strata_1 and Strata_2. In this case, Strata_2 is encapsulated within Strata_1. At run time, Strata_2 translates the application to its software cache, SC_2. Strata_1 translates Strata_2 and SC_2 into SC_1. Because the code resident in SC_2 does not execute directly, it can be encrypted.

feature is important to note as it indicates that the code residing in the software caches of inner[2] PVMs (such as Strata_2) do not execute directly. Therefore, they can be transformed in a manner that thwarts analysis (e.g., encryption).

IV. Evaluation

One of the major goals of nested virtualization is to increase the obfuscation of software cache containing translated code. To evaluate this feature, we analyzed the software caches of the N-PVMs.

A. Protection Evaluation

As we mentioned, code resident in the software cache of any N-PVM that is not directly in contact with the native platform will be translated by another N-PVM. As a means of thwarting analysis, the contents of such an inner-level software cache can be encrypted. In such a case, the decryption key must be possessed by any PVM that is translating this code. In the current example, Strata_1 responsible for mediating Strata_2, and must possess the corresponding decryption key. This feature can be extended to multiple layers of virtualization, as long as the decryption keys are shared appropriately. Thus, if a PVM at level n controls the execution of a PVM at level $n+1$, it must possess the decryption key for the encrypted code residing in the software cache of the PVM at level $n+1$.

Care must be taken to ensure the encryption algorithm is robust, and the key is not easy to extract. White-box cryptography is often used to protect the encryption key [12].

We now focus on the software cache of the PVM at the outermost level, i.e., the PVM that executes on the underlying hardware. In our example, this PVM is represented by

[2]In this discussion, positions of the virtualization layer are described relative to the application. The application is located at level n, the innermost PVM layer at level $n-1$. Any PVM that is interpreted directly by the hardware is level 1.

Strata_1. Figure 3 illustrates its software cache as measured on the `181.mcf` benchmark. As the figure shows, there is interleaving between the code from these two components. There is no clear demarcation between the code of Strata_2 and the application.

Interleaving code from the application and the PVM provides greater entropy and leads to greater obfuscation. It is more difficult for the adversary to distinguish the application code from the PVM code. We used the following approach to obtain some measure of the entropy of the system. Each basic block that originated from the application was assigned a bit value of '0', and each basic block from Strata_2 was assigned a value of '1'. A string was then obtained based on the layout of the software cache, and compressed using the `LZ78` algorithm.

In the case of an application protected by a single PVM, the compression ratio was 149. In the case of the nested configuration, the compression ratio was 15.63. These results are encouraging from the viewpoint of the software defender. With N-PVM configurations, the software caches will possess more entropy, thereby making it harder for the adversary to extract meaningful information.

Next we demonstrate that the high-level information (e.g., the CFG) is also more obfuscated, further thwarting the adversary from successfully reverse-engineering the application.

To validate our claim of increased CFG complexity, we reviewed past work and identified a metric, Cyclomatic Complexity, for measuring complexity of program graphs. Cyclomatic Complexity (CC) was designed by Thomas McCabe, and is used to indicate the complexity of graphs. It measures the number of linearly independent paths through an application's code [13]. In the context of reverse engineering, a higher value of CC implies that there are more program paths that need to be analyzed. Consequently, more effort is required from the adversary. CC has been used previously in the field of program obfuscation [14], [15]. McCabe, *et al.* defined the Cyclomatic Number (CN), for a undirected graph G, as:

$$CN(G) = e(G) - n(G) + 2 * p(G) \quad (1)$$

where $e(G)$ represents the number of edges of the graph, $n(G)$ denotes the number of nodes in the graph, and $p(G)$ denotes the number of exit nodes in the graph [13].

Our experiment consisted of comparing the cyclomatic complexities of the CFGs obtained from the software caches in two different configurations; the application running under control of a single PVM, and the application running under control of a N-PVM configuration (i.e., the application running under Strata_2, which runs under Strata_1). In the nested configuration, we only consider the software cache of Strata_1. We used the benchmarks in the SPEC CPU 2000 suite and used the test inputs to facilitate collection of the CFGs.

The CFG is built from the executable code located in the software cache. When control exits the software cache and enters the SDT library, processing of the CFG stops. When control returns to the software cache, a new CFG component is started. So the dynamic CFG consist of several disconnected components representing the different translated blocks. In the case of the nested configuration, only the the software cache of Strata_1 (SC_1) is considered. This software cache contains code translated from the inner PVM (Strata_2), as well as code originating from the application.

Table I displays the cyclomatic complexity for some of the benchmarks, under the two scenarios. As the data in the table indicates, N-PVM configuration has more independent paths in its CFG, compared to the case where only a single PVM is used. On closer examination of Equation 1, we observed that the increase in CC in the nested configuration was mainly triggered by the increase in the number of exit nodes (denoted by $p(G)$ in Equation 1, which has twice the weight of the other parameters). This higher value of $p(G)$ implies that there are more disconnected components in the software cache of the outer PVM (Strata_1). An adversary would have to collate all these extra components to successfully reverse engineer the application.

The experiment results indicate that N-PVMs have the potential of increasing the obfuscation of protected applications. Simple nesting of PVMs yields CFGs that are significantly more complex. With more elaborate partitioning and utilization of numerous N-PVMs, foundations can be laid for robust program protection. An adversary would have to expend significantly more effort to successfully obtain relevant information.

This section provided some insight into the protection properties of N-PVMs. However, to be used in practice, the run-time overhead of N-PVMs must be tolerable. In this section, we discuss the performance implications of N-PVMs, and suggest techniques to alleviate the overhead.

B. Performance Evaluation

The performance evaluation was performed using the SPEC CPU 2000 benchmarks. The experiments were carried out on a 32-bit AMD Athlon processor, running Ubuntu 12.04. The code was compiled using the `gcc-2.95` compiler, and the protected package was created using Diablo.

The performance overhead was observed to be 35X over native execution, on average. This high overhead is due to the overhead associated with software dynamic translation of *self-modifying* code. To understand the source of the overhead, it is necessary to review how software dynamic translators handle branches.

The SDT translates and caches instructions from the application one at a time until a control transfer instruction is encountered. If the target block (also called a *successor* block) of the transfer instruction has previously been translated and cached, the SDT will append a direct transfer instruction to that block. If the target is absent, the SDT will append a sequence of instructions that transfer control back to itself so that it can translate the code located at the target address. This sequence of instructions is known as a *trampoline*. Each trampoline is associated with a target address.

When the target application block actually appears in the software cache the corresponding trampoline is rewritten with a `jump` instruction, that transfers control directly to the

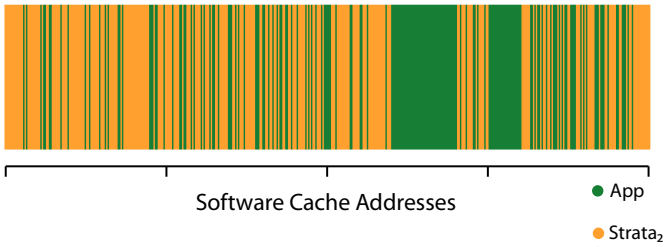

Fig. 3. This figure illustrates the software cache of $Strata_1$ as measured on the `181.mcf` benchmark. This software cache contains the code from $Strata_2$, and its software cache SC_2, which is basically translated application code. As can be seen from the figure, the code from the components are interleaved, to provide better obfuscation.

TABLE I
CYCLOMATIC COMPLEXITY OF THE DYNAMIC CFGS OBTAINED FROM THE SOFTWARE CACHES, WHEN RUN UNDER A SINGLE PVM, AND A N-PVM CONFIGURATION COMPRISING TWO PVMS.

Benchmark	CC for Single PVM	CC for Nested PVMs	Increase from Original
176.gcc	1,604	80,109	4,894%
181.mcf	351	9,828	2,701%
256.perlbmk	803	32,903	3,997%
179.art	181	5,130	2,734%

translated target block, without invoking the SDT again. This instruction rewriting process is known as *patching*. Scott *et al.* provides a more complete discussion for the interested reader [9].

Patching is essentially a special kind of self-modifiying code. It works well where there is only a single level of dynamic translation as the update is made to the cache from which the hardware will fetch instructions. Modern hardware handles such situations well, although there are still some execution penalties because cache lines must be flushed.

Currently, SDTs handle self-modifying code by flushing the entire software cache, and continuing translation at the next application block scheduled to be executed. In our N-PVM case study, $Strata_1$ obtains control at start up, and begins translating $Strata_2$'s code to its software cache, SC_1. Consequently, control is transferred to the translated code corresponding to $Strata_2$. This code translates the application to the software cache, SC_2, and appends a trampoline to those blocks whose target successors have not been translated. Consequently, $Strata_1$ again obtains control and copies the instructions from SC_2 to SC_1.

When the translated code for $Strata_2$ patches a trampoline in its software cache (SC_2), it causes $Strata_1$ to flush SC_1 entirely. After the flush, $Strata_1$ must translate a significant portion of $Strata_2$ (such as the initialization parts), before any code corresponding to the application can be translated and executed.

The simple mechanism of handling self-modifying code in SDTs (i.e, flush the entire software cache) is expensive and leads to high overheads in N-PVMs. Because the software cache flushes are most often caused by the patching of trampolines, we devised a novel mechanism, called *super-patching*, that alleviates some of the performance overheads associated with N-PVMs.

As previously described, an update to the software cache of $Strata_2$ via a patch operation, causes the entire software cache of $Strata_1$ to be flushed. Instead of flushing, a simple optimization involves propagating the patch in the software cache of $Strata_2$ into the software cache of $Strata_1$. When any trampoline is being patched to its target block (TB) in SC_2, $Strata_2$ sends that information about the patch and the TB to $Strata_1$. $Strata_1$ no longer flushes the cache but stores this information for later processing. When the TB is translated from SC_1 to SC_2, $Strata_1$ artificially patches the *translated* trampoline to the translated TB in its software cache. This mechanism is called super-patching, because the patching in SC_1 triggers this patch in SC_2. With this mechanism, the extraneous flush has been removed.

Figure 4 compares the overhead for an application running under a single Strata instance, and the N-PVM. Due to the reduction in unwanted software cache flushes, the total overhead is now 70% over native. One of the reasons for the high overhead pertains to indirect branches. Applications with a high occurrence of indirect branches are known to cause performance issues in SDTs [16]. Because the SDT itself uses indirect branches during run time, adding the second layer causes the overhead to increase significantly. This amplification is specially visible in benchmarks such as `176.gcc` and `253.perlbmk`, which also have a high occurrence of indirect branches.

V. RELATED WORK

There is a large amount of work in the area of software protection. Collberg and Nagra provide an excellent introduction to the area [17]. Recently there has been interest

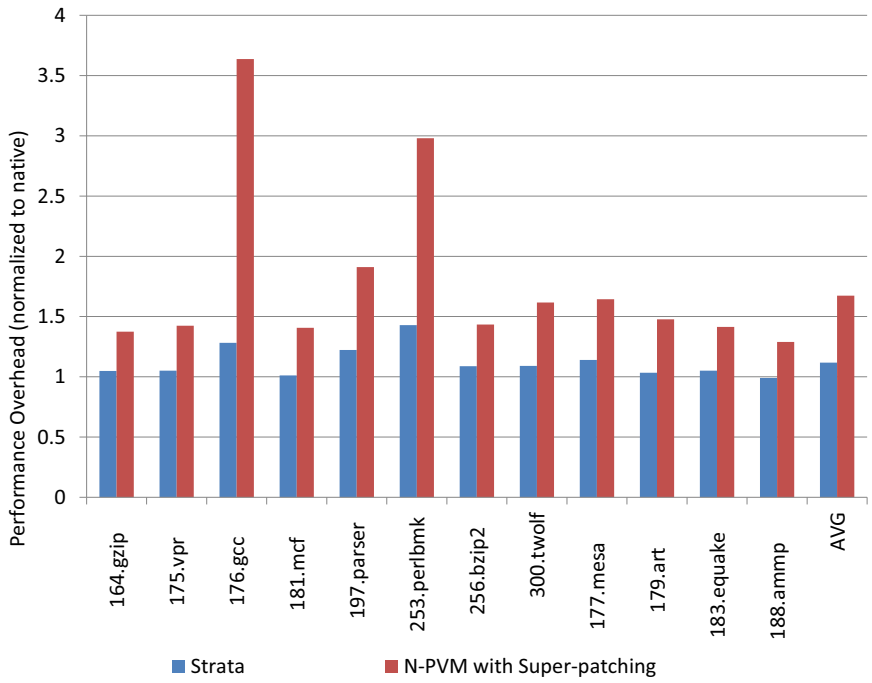

Fig. 4. Performance overhead for super-patching in the nested PVM configuration. The performance overhead of baseline Strata is also provided for comparison. On average, super-patching reduces the overhead of nested PVMs to 70% over native execution.

in using virtual machine for software protection. One of the most early works in the area was Proteus [18]. Despite the run-time overhead (50x to 3500x), the work demonstrated the promise of virtualization to provide software protection. Subsequent work has shown that software protection can be made practical through virtualization [1]–[4], [10]. System-level virtual machines have also been proposed as a means to provide software protection. The Terra system includes a trusted virtual machine monitor which can be used to create close-box platforms that allow the software stack to be tailored to provide protection [19]. The system does assume hardware support to validate the software stack. Chen *et al.* discuss Overshadow [20]. Overshadow cryptographically isolates an application for the guest OS using a VM. Thus, the system offers another layer of tamper resistance, even in the case of total OS compromise.

As virtualization techniques became a viable approach to protect software (being used both for legitimate purposes and to protect malware), researchers began to investigate techniques for reverse engineering code protected via virtualization. Some of the early work in this area was done by Rolf Rolles [21]. This paper discusses how to defeat VMProtect [2]. Sharif *et al.* also investigated techniques to reverse virtualization-based obfuscation techniques. The approaches of both Rolles and Sharif *et al.* attempt to reverse engineer the VM (or interpreter) and then attempt to reverse engineer the byte code program. A different approach was proposed by Coogan *et al.* [22]. They discussed an approach which focuses on identifying the flow of values to system call instructions used by malware thereby separating the application code from the virtual machine code. All of these works assume that the interpreters do not perform any dynamic translations or generate new code, making control and data flow analysis easier to perform on the generate traces.

VI. Conclusions

This paper has introduced the concept of nested process-level virtual machines (N-PVMs). The preliminary results show that N-PVMs can strengthen protection as measured by common software protection metrics such as entropy and cyclometric complexity. Complete protection of an application by nested N-PVMs incurs high run-time overhead, which indicates N-PVMs must be applied carefully to critically important sections of an application. Further study is needed on a set of real applications with assets that must be protected to fully evaluate the run-time overhead and strength of protection of N-PVMs.

VII. Acknowledgments

This research was supported by National Science Foundation (NSF) grant CNS-0716446, Army Research Office (ARO) grant W911-10-0131, and Air Force Research Laboratory

(AFRL) contract FA8750-13-2-0096. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF, AFRL, DoD, or the U.S. Government.

REFERENCES

[1] Oreans Technologies, "Themida," http://oreans.com/themida.php, 2009.

[2] VMProtect Software, "VMProtect," http://vmpsoft.com/, 2008.

[3] S. Ghosh, J. D. Hiser, and J. W. Davidson, "A secure and robust approach to software tamper resistance," in *Proceedings of the 12th International Conference on Information Hiding*, ser. IH'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 33–47. [Online]. Available: http://dl.acm.org/citation.cfm?id=1929304.1929307

[4] Oreons Technology, "CodeVirtualizer," http://oreans.com/codevirtualizer.php, 2009.

[5] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers, "Secure program partitioning," *ACM Trans. Comput. Syst.*, vol. 20, no. 3, pp. 283–328, Aug. 2002. [Online]. Available: http://doi.acm.org/10.1145/566340.566343

[6] S. H. K. Narayanan, M. Kandemir, and R. Brooks, "Performance aware secure code partitioning," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '07. San Jose, CA, USA: EDA Consortium, 2007, pp. 1122–1127. [Online]. Available: http://dl.acm.org/citation.cfm?id=1266366.1266609

[7] D. Søndergaard, C. W. Probst, C. D. Jensen, and R. R. Hansen, "Program partitioning using dynamic trust models," in *Proceedings of the 4th International Conference on Formal Aspects in Security and Trust*, ser. FAST'06. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 170–184. [Online]. Available: http://dl.acm.org/citation.cfm?id=1777688.1777700

[8] T. Zhang, S. Pande, A. dos Santos, and F. J. Bruecklmayr, "Leakage-proof program partitioning," in *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '02. New York, NY, USA: ACM, 2002, pp. 136–145. [Online]. Available: http://doi.acm.org/10.1145/581630.581651

[9] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa, "Retargetable and reconfigurable software dynamic translation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 36–47. [Online]. Available: http://dl.acm.org/citation.cfm?id=776261.776265

[10] S. Ghosh, J. D. Hiser, and J. W. Davidson, "What's the pointisa?" in *Proceedings of the 2Nd ACM Workshop on Information Hiding and Multimedia Security*, ser. IH&MMSec '14. New York, NY, USA: ACM, 2014, pp. 23–34. [Online]. Available: http://doi.acm.org/10.1145/2600918.2600928

[11] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere, "Diablo: a reliable, retargetable and extensible link-time rewriting framework," in *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology*, Dec 2005, pp. 7–12.

[12] S. Chow, P. A. Eisen, H. Johnson, and P. C. v. Oorschot, "White-box cryptography and an AES implementation," in *SAC '02: Revised Papers from the 9th Annual International Workshop on Selected Areas in Cryptography*. London, UK: Springer-Verlag, 2003, pp. 250–270.

[13] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, pp. 308–320, Jul. 1976. [Online]. Available: http://dx.doi.org/10.1109/TSE.1976.233837

[14] B. Anckaert, M. Madou, B. De Sutter, B. De Bus, K. De Bosschere, and B. Preneel, "Program obfuscation: A quantitative approach," in *Proceedings of the 2007 ACM Workshop on Quality of Protection*, ser. QoP '07. New York, NY, USA: ACM, 2007, pp. 15–20. [Online]. Available: http://doi.acm.org/10.1145/1314257.1314263

[15] M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella, "Towards experimental evaluation of code obfuscation techniques," in *QoP '08: Proceedings of the 4th ACM Workshop on Quality of Protection*. New York, NY, USA: ACM, 2008, pp. 39–46. [Online]. Available: http://doi.acm.org/10.1145/1456362.1456371

[16] J. D. Hiser, D. Williams, A. Filipi, J. W. Davidson, and B. R. Childers, "Evaluating fragment construction policies for sdt systems," in *Proceedings of the 2nd International Conference on Virtual Execution Environments*, ser. VEE '06. New York, NY, USA: ACM, 2006, pp. 122–132. [Online]. Available: http://doi.acm.org/10.1145/1134760.1134778

[17] C. Collberg and J. Nagra, *Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection*. Pearson Education, 2009.

[18] B. Anckaert, M. Jakubowski, and R. Venkatesan, "Proteus: Virtualization for diversified tamper-resistance," in *Proceedings of the ACM Workshop on Digital Rights Management*, ser. DRM '06. New York, NY, USA: ACM, 2006, pp. 47–58. [Online]. Available: http://doi.acm.org/10.1145/1179509.1179521

[19] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 193–206. [Online]. Available: http://doi.acm.org/10.1145/945445.945464

[20] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 2–13. [Online]. Available: http://doi.acm.org/10.1145/1346281.1346284

[21] R. Rolles, "Unpacking virtualization obfuscators," in *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, ser. WOOT'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855876.1855877

[22] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of virtualization-obfuscated software: A semantics-based approach," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 275–284. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046739