A Framework for Multi-Variant Execution Environment

Wang Junchao\textsuperscript{a} Liu Zhenwu\textsuperscript{b} Liu Hao\textsuperscript{c} Pang Jianmin\textsuperscript{d}

State key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, China
\textsuperscript{a}wangjunchao11@126.com \textsuperscript{b}mjandlzw@qq.com \textsuperscript{c}liuhao_work@qq.com \textsuperscript{d}jianmin_pang@126.com

Abstract. Software diversity has been proven to be an effective approach to enhance system security. To make the best of the advantage brought by software diversity, a multi-variant execution environment is needed. However, although some MVEEs have been proposed, most of them are either too simple or only focusing one aspect which limits their widely adoption in industry. In this paper we propose a framework for multi-variant execution environment to enhance the security of software systems. The framework addresses different aspects when implementing a MVEE and can help to make the best of software diversity to enhance the system’s security.

1. INTRODUCTION
Cyber security is becoming increasing important in recent years. More and more smart devices are accessing the Internet. The prevalence of smart devices brings us quite a lot of conveniences in our daily life and work; however, security issues emerge at the same time. Security issues exist in the whole lifecycle of the software development phases, ranging from the design to the software’s deployment and execution. For instance, a software may have design flaws which can be exploited by attackers. To cope with such problem, formal methods or model checking approaches are applied. A software’s implementation may also have vulnerabilities due to the inappropriate or careless usage of certain functions. Code auditing is a promising approach for checking whether there are any potential vulnerabilities inside a software. For web applications, DDoS attack is a comparatively big threat. Firewalls are commonly used to deal with such issue. As we can see from above, nowadays a corresponding defending approach is designed towards a certain type of threat.

The cyber system is becoming increasingly complex. Static and predictable defensive measures can be applied to detect and defend most malicious activities. However, attacking techniques are constantly evolving. For instance, 0-day vulnerabilities are still increasingly found, which serves as a main threat in nowadays computing systems. Traditional defenses are no longer sufficient to effectively combat the newly designed attack techniques. At the same time, due to critical software security patches are often difficult to deploy in time, network attackers have certain strategic advantages. Therefore, we need to develop proactive defense technology to deal with unknown vulnerabilities and to maximize the security benefits for defenders. A system is always vulnerable in a certain sense. Essence of encryption is to increase the difficulties of attackers to a system. MTD (Moving Target Defense) technology \cite{1} and mimic defending technology \cite{2} both share similar ideas. The idea of MTD is based on the assumption that an attack to a software may not succeed on its diversified version. Mimic defense technology is an active defense technology, which brings heterogeneous redundancy to software
deployments, can tolerate operational errors and detect threat-to-security attacks when running multiple
versions of programs in parallel [2]. Mimic security addresses mitigating the attack during run-time of a
software by changing the software’s architecture, executables to increase the difficulties of an attacker.

Software diversity is a basic research topic which serves as the foundation of MTD and mimic defense technology. It has been proven to be an effective measure against known and unknown attacks. Software diversity guarantees that at least one variant can provide normal service when deploying more than one variants at a time. An attack can trigger an alarm because different behaviors in the variants can be detected, thus providing probabilistic security. Research on software diversity mainly focuses on generating different variants and evaluate whether a transformation is effective for coping with an attack. For instance, by simply inserting NOP (no-operation) instruction during compilation process, Homescu et al. [3] prove that code-reuse attacks can be effectively thwarted with low performance overhead.

To take the best advantage of software diversity, how the diversified variants are managed, monitored and scheduled needs a execution framework. Thus, the concept of MVEE (Multi-Variant Execution Environment) is proposed [4]. However, although different MVEE frameworks are proposed in recent years [5], [6], the frameworks are either too simple or cannot be directly applied in a complex application with critical QoS requirements. Therefore, in this paper we propose a comprehensive MVEE framework to address the different aspects when applying the idea of software diversity.

2. RELATED WORK
The MVEE framework is mainly related to the MVEE frameworks or architectures and how the variants are generated. Thus, we summarise existing works related to the above two topics.

2.1 N-variants system and MVEE
The idea of N-variants system can originates from Nversion programming. In 1977, Chen et al. proposed N-version programming to develop the same software with different development groups so that multiple versions of software can be produced without the same faults [8]. However, Nversion programming is difficult to be applied because usually a software development requires more than two teams. The idea is applied to enhance the fault tolerance of a software. In 1993, Cohen proposes “program evolution” where programs can evolve into semantically equivalent versions to defend automated attacks on operating systems [7]. In 2006, Cox et al. propose a simple N-variant system framework [9] with two basic components. A polygrapher takes input from the client and spreads to all variants. Abnormal behavior of variants can be detected by a monitor. However, the framework is generally too simple for handling multiple variants during the variants’ execution. Moreover, it neglects the users’ requirements on QoS properties such as performance and cost which hinders its wide adoption.

To make the multivariant systems practical, Franz et al. focus on designing applicable multicompiler to generate multiple variants from the same program source and check the behavior of variants in lockstep [10]. The MVX architecture is relatively mature compared with previous proposed MVEE architecture [6]. Key component of the MVX is the variant manager. Namespace manager ensures the same information transmitted to variants. Detector compares the divergence between the inputs and outputs of variants. Security manager generates security policies to guarantee the tradeoff between security and performance. Variants generator diversifies an application with different transformation techniques. As we can see from above architectures, the MVEE system is becoming increasingly complex. To the best of our knowledge, no existing work before this paper addresses the different aspects when designing the architecture of a MVEE.

2.2 Variants generation
Cohen demonstrated several basic program evolution techniques such as instruction replacement, variable substitution, instruction reordering and garbage code insertion [7]. Stack frame randomization [11] can defend stack-based attacks with variable reordering and stack frame padding to diversify the
binaries. Andrei et al. applied profile-guided optimization to reduce the performance overhead of software diversity [3]. They applied the garbage insertion diversification technique and inserted the NOP instruction during the compilation process. They proved that even with high performance overhead randomization techniques can become effectively practical with the profile-guided diversification. Code layout randomization can increase the cost of code reuse attack. Giuffrida et al. proposed the address space randomization (ASR) to enhance the OS-level security [12]. Modern operating systems provide Address Layout Randomization (ASLR), which randomizes the base addresses of stacks, heaps, dynamic libraries etc. ASLR mainly makes modifications on the base address of program objects. However, it does not provide diversity inside the sections. Program obfuscation techniques can help to generate the diverse variants [13]. Xu et al. classify the program obfuscation approaches into code-oriented obfuscation and model-oriented obfuscation. Code-oriented obfuscation techniques focus on applying transformations to programs from software engineering perspective. Comparatively, model-oriented obfuscation is more theoretical by analysing how much information can be hidden with a certain obfuscation. Specifically, code-oriented obfuscation can be implemented with several different approaches. Preventive obfuscation can increase the difficulty for decompilers [13]. Layout obfuscation can help to change instructions’ orders or obfuscates the identifiers of variables or classes [13]. Control flow obfuscation can be implemented by inserting opaque predicates to increase the complexity of a program [14]. Data obfuscation obscures data by encoding [15] or encrypting data inside a program [16]. Plymorphic obfuscation is usually applied by attackers to hide malware from anti-virus detection. Randomization mechanisms such as data pointer randomization can generate multiple obfuscated variants simultaneously [17]. As we can see from above descriptions, there exists many software transformation techniques. These techniques mainly focus on how the variants are generated. However, how to make the best of the variants to secure a system still needs the support of MVEE framework.

3. FRAMEWORK

Fig. 1 shows the MVEE framework proposed in this paper. The framework is mainly consisted of five components: knowledge base, decision module, infrastructure manager, monitor and variants controller. Knowledge base provides models, rules and algorithms to other components in the MVEE framework. Decision module helps to decide the infrastructures needed for the application so that the application’s QoS requirements are guaranteed. Infrastructure manager manages the provisioned infrastructures. Monitoring module monitors the states of the variants and checks whether there exists any abnormal behaviour of variants. Variants controller can help to control how the variants are generated and transformed. In the following subsection we will elaborate on each module.

3.1 Application description

Nowadays most applications are consisted of more than one components. For instance, a web application usually consists a web server, a database management system etc [18]. Each component can be deployed on distributed clusters. Management of the distributed components onto computing/storage infrastructures is often sophisticated to guarantee the services’ persistance. Moreover, the application can have multiple different QoS requirements such as performance, cost and security. The QoS requirements are not always stimulative with each other. For instance, the higher performance of the infrastructures can improve the system’s resistance against DoS/DDoS attacks. However, the cost is correspondingly higher.

3.2 Proactive Decision Module

Some application developers mainly focus on the functional requirements of the system. Specifically, an application should produce the expected outputs and perform the “right” behaviour specified. However, nowadays non-functional QoS requirements need to be met apart from the functional requirements of a system. For instance, some real time industrial applications should finish the data processing within the deadline specified by the user. The makespan of real time applications is a non-
functional QoS parameter which should be finished within a specified deadline. Other QoS parameters include monetary cost and security etc. In this paper we mainly focus on the security of an application.

![Diagram]

Figure 1 Our proposed MVEE Framework

There are different traditional ways to enhance the security of an application. For instance, the security of an application intuitively relies on the security and reliability of the computing and storage infrastructures it is executed on. When the infrastructures crash, the application will crash at the same time. Secondly, an application’s security can be enhanced with code auditing services checking whether there are any potential vulnerabilities inside an application. Thirdly, an application’s security can also be improved by facilitating with security defense systems that is independent to the application such as firewalls, IDS etc. The second and third security enhancing approaches need more investment by using or renting related services. However, provisioning the appropriate infrastructures and services for an application can be quite difficult because the different QoS aspects should be carefully considered and tradeoffs between these aspects should be made. In this paper, the proactive decision module can help to guarantee the systems security and other QoS requirements by provisioning the right infrastructure and services.

Different from traditional security enhancing approaches, the mimic defense technology argues that an application’s security can be guaranteed with certain probability by deploying multiple functional equivalent variants. The first problem before deployment of an application is to decide the right number of variants it needed. In our previous work [19], we propose a software assignment algorithm which can help the user decide the right number of variants for an application defending worm-like attacks.

3.3 Variants Controller

The variants controller can help to manage the polymorphic executable binaries. The variants controller manages a variants pool. It can replace the variants pool with newly generated binaries. Variants pool
can be implemented with a distributed file system. Moreover, it can help to determine where to place the executables onto the infrastructures. Appropriate placement of variants can help to reduce the variant transmission time. It can also receive requests from the monitor and sends a COTS-like software.

Generally a functional equivalent executable can be generated with the variant generator in two modes: on-line and off-line modes. For off-line modes, the executables can be diversified when providing the source code or not. When the application allows variants manager access to the source code of the application, the multicomplier [10] can help to generate the diversified binaries. When the user doesn’t want to share his source code and only provide executable binaries, reverse engineering can be applied such as instrumenting garbage codes to the binaries to change its superficial characteristics. Moreover, IR can be reverse-engineered. Similar transformations used by multicomplier can also be applied to make modifications and regenerate the executables. For on-line modes, the ASLR and stack reversing techniques [13] can be applied.

The controller also changes the variants dynamically during runtime. This context switching should at least guarantee the normal behavior of the variants. Before quitting, a variant should finish processing all the previous requests. Moreover, it should broadcast to other components that it is quitting. When a new substitute is ready, the controller should help broadcast its arrival.

3.4 Infrastructure manager
The infrastructures provide basic computing/storage services to applications. Infrastructures can be delivered in the cloud model. Generally the cloud provides services in three ways: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), Software-as-a-Service (SaaS). IaaS directly provide infrastructures to users. PaaS and SaaS directly provide execution runtime, database, web server, email etc. to users. This makes the application variants generation less flexible. Thus, in this paper we choose the infrastructure service to facilitate the mimic software stack.

After starting the infrastructure, the infrastructure manager should help to manage the underlying infrastructures. It should also make changes to the infrastructures when the workload changes or abnormal behaviours happen etc.

3.5 Monitor
Monitor can help to detect the state of the variants. The monitor should support both the proactive mode and the passive mode. Different checkpoint strategies can be applied. For instance, MvAmor checks the output of each system call [6], comparing whether the outputs of each variant are equal. We can monitor the state of the application by sending checking packets proactively or listening to the components’ request. A synchronizer and checker are needed. The synchronizer inside the monitor synchronizes the states of variants and infrastructure within a certain checkpoint. Synchronized information serves as the basis for abnormal behavior detector.

The monitor can also communicate with the controller, requesting the controller to change the state of the variants seamlessly without interrupting the normal behavior of the variants.

The monitor’s security can be enabled through virtualization boxes like Dune[6]). Moreover, since the monitor’s function is limited, model checking/formal method can help to guarantee the monitor’s security. Another extreme solution to a monitor is to use FPGA or ASIC to implement its key function.

3.6 Scheduler
The scheduler helps to schedule the variants on to the infrastructures. Before starting the application, the components should be determined where to place. Moreover, the scheduler should dynamically schedule the application’s components onto the infrastructures when the workload or unexpected attacks happen. When there are changes happened to the infrastructures, the scheduler should also schedule the components onto the infrastructures.

The scheduler can also help to mitigate a variant from a VM to another. Sometimes, the whole architecture of the application needs to change from a case to another. For instance, in a dynamic SDN (Software-defined Networking) environment, the controller may be mitigated from a network node to
another. In such a case, the scheduler should make response and deploy the components onto the scheduled infrastructures.

3.7 Knowledge Base
The knowledge base is consisted of models, rules and specific data. Different applications may have different characteristics. For instance, the email server is quite different from a website although all their services are to serve clients. Thus, the knowledge base should store different models, algorithms when making decisions to these infrastructures.

Known attack models and methods can be stored in the knowledge base. Attack detection methods are still prevailing techniques in current software systems. The abnormal behaviour detector of the monitor relies on the rules and models in the knowledge base to help determine whether there are potential attacks.

4. EXPERIMENTS
In this part, we mainly evaluate the effectiveness of our proposed framework based on two common vulnerabilities. The first vulnerability is a “house of einherjar”. By assigning the return address of “malloc” to any chunk, we can abuse the merging of “free”. Since the monitor of our proposed framework can detect the different output of one variant with another, we can easily identify and defend such an attack.

Moreover, we can also defend the “Use After Free” vulnerability when re-allocating memories.

5. CONCLUSION AND FUTURE WORKS
In this paper we propose a MVEE framework to compensate the urgent need for software diversity deployment in industry. The framework is consisted of five key components: variants controller controlling how and when to generate a new variant, scheduler determining how the variants and infrastructures scheduled, monitor, knowledge base and decision module.

In our future work we will propose explicit algorithms, models and implementations for each module in the framework.

ACKNOWLEDGMENT
This research is supported by National Key Research and Development Program of China (2017YFB0803202).

REFERENCES
[1] S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, Moving target defense: creating asymmetric uncertainty for cyber threats, vol. 54. Springer Science & Business Media, 2011.
[2] J. Wu, “Introduction to cyberspace mimic defense,” Telecommunication Science, vol. 12, 2017.
[3] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, “Profile-guided automated software diversity,” in Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 1–11, IEEE Computer Society, 2013.
[4] B. Salamat, Multi-Variant Execution: Run-Time Defense against Malicious Code Injection Attacks DISSERTATION. PhD thesis, University of California, Irvine, 2009.
[5] S. Volekaert, B. Coppens, and B. De Sutter, “Cloning your gadgets: Complete rop attack immunity with multi-variant execution,” IEEE Transactions on Dependable and Secure Computing, vol. 13, no. 4, pp. 437–450, 2016.
[6] K. Koning, H. Bos, and C. Giuffrida, “Secure and efficient multi-variant execution using hardware-assisted process virtualization,” in 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 431–442, IEEE, 2016.
[7] F. B. Cohen, “Operating system protection through program evolution.,” Computers & Security, vol. 12, no. 6, pp. 565–584, 1993.
[8] L. Chen and A. Avizienis, “On the implementation of n-version programming for software fault tolerance during program execution,” in International Computer Software and Applications Conference (COMPSAC), 1977.

[9] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, “N-variant systems: A secretless framework for security through diversity,” in USENIX Security Symposium, pp. 105–120, 2006.

[10] M. Franz, “Making multivariant programming practical and inexpensive,” IEEE Security & Privacy, vol. 16, no. 3, pp. 90–94, 2018.

[11] S. Forrest, A. Somayaji, and D. H. Ackley, “Building diverse computer systems,” in Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133), pp. 67–72, IEEE, 1997.

[12] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, “Enhanced operating system security through efficient and fine-grained address space randomization,” in Presented as part of the 21st [USENIX] Security Symposium ([USENIX] Security 12), pp. 475–490, 2012.

[13] H. Xu, Y. Zhou, Y. Kang, and M. R. Lyu, “On secure and usable program obfuscation: A survey,” arXiv preprint arXiv:1710.01139, 2017.

[14] C. Collberg, C. Thomborson, and D. Low, “A taxonomy of obfuscating transformations,” tech. rep., Department of Computer Science, The University of Auckland, New Zealand, 1997.

[15] L. Ertaul and S. Venkatesh, “Novel obfuscation algorithms for software security,” in Proceedings of the 2005 International Conference on Software Engineering Research and Practice, SERP, vol. 5, Citeseer, 2005.

[16] A. Kovacheva, “Efficient code obfuscation for android,” in International Conference on Advances in Information Technology, pp. 104–119, Springer, 2013.

[17] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz, “It’s a trap: Table randomization and protection against function-reuse attacks,” in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 243–255, ACM, 2015.

[18] S. Allier, O. Barais, B. Baudry, J. Bourcier, E. Daubert, F. Fleurey, M. Monperrus, H. Song, and M. Tricoire, “Multitier diversification in web-based software applications,” IEEE Software, vol. 32, no. 1, pp. 83–90, 2014.

[19] Wang Junchao, Liu Hao, etc. “Topology-aware software assignment in networked systems,” in 4th International Conference on Electronic Technology and Information Science, 2019.