The MINESTRONE Architecture Combining Static and Dynamic Analysis Techniques for Software Security

Angelos D. Keromytis
Columbia U.
angelos@cs.columbia.edu

Salvatore J. Stolfo
Columbia U.
sal@cs.columbia.edu

Junfeng Yang
Columbia U.
junfeng@cs.columbia.edu

Angelos Stavrou
George Mason U.
astavrou@gmu.edu

Anup Ghosh
George Mason U.
aghosh@gmu.edu

Dawson Engler
Stanford U.
engler@csl.stanford.edu

Marc Dacier
Symantec Research Labs
marc_dacier@symantec.com

Matthew Elder
Symantec Research Labs
matthew_elder@symantec.com

Darrell Kienzle
Symantec Research Labs
darrell_kienzle@symantec.com

I. Problem Statement

We present MINESTRONE, a novel architecture that integrates static analysis, dynamic confinement, and code diversification techniques to enable the identification, mitigation and containment of a large class of software vulnerabilities in third-party software. Our initial focus is on software written in C and C++; however, many of our techniques are equally applicable to binary-only environments (but are not always as efficient or as effective) and for vulnerabilities that are not specific to these languages. Our system seeks to enable the *immediate* deployment of new software (*e.g.,* a new release of an open-source project) and the protection of already deployed (legacy) software by transparently inserting extensive security instrumentation, while leveraging concurrent program analysis, potentially aided by runtime data gleaned from profiling actual use of the software, to gradually reduce the performance cost of the instrumentation by allowing selective removal or refinement. Artificial diversification techniques are used both as confinement mechanisms and for fault-tolerance purposes. To minimize the performance impact, we are leveraging multicore hardware or (when unavailable) remote servers that enable quick identification of likely compromise. To cover the widest possible range of systems, we require no specific hardware or operating system features, although we intend to take advantage of such features where available to improve both runtime performance and vulnerability coverage.

The fundamental problem being addressed in this project – finding vulnerabilities in software — is being addressed in the commercial marketplace today by a combination of tools and expertise. Today, companies such as Coverity, Klocwork, Ounce Labs, and Fortify have developed sophisticated source code analyzers that analyze C/C++ and Java code for known vulnerabilities. Other tools such as ITS4, RATS, and cppcheck also provide vulnerabilities with varying degrees of effectiveness. Most of these products are state-of-the-art releases of software research and represent the best software vulnerability analysis has to offer today. One common attribute of these tools is that they produce a large number of false positives—warnings of potential vulnerabilities that often are not true. As such, they require software security expertise—a need that is met by commercial consulting offerings, some by the tool vendors themselves. Our ultimate goal is to take advantage of these and other analysis techniques without having to expose users, programmers or administrators to their output.

To address shortcomings in software vulnerability analysis, software fault isolation [1] or confinement techniques [2] can be used to limit the effects of residual software vulnerabilities. Software fault isolation techniques can be used to confine the bounds of program execution. Failure oblivious computing [3] is a set of software techniques to continue executing even in the presence of faults. Similarly, error virtualization uses a program's native error handling routines to mask faults while returning the program to a known safe state after an error [4]. Almost all confinement approaches require program instrumentation that imposes additional instruction execution time, which means increased (potentially substantial) overhead. The finer-grained the instrumentation, the finer the containment and the more overhead is required. The coarser the instrumentation, on the other hand, the lower the overhead, but the higher the likelihood for missed or unhandled faults. We will use code instrumentation to implement software fault isolation and process-level confinement to limit malicious software behavior. We will leverage our analysis to remove or refine this instrumentation so as to minimize or eliminate its performance impact when possible.

II. Research Direction

The overall MINESTRONE architecture and system workflow is shown in Figure 1. The intellectual core and novelty of our approach revolves around establishing and leveraging a feedback loop among a hardened production system, program analysis, and diversification. The key idea is that static analysis will allow us to target the instrumentation, while runtime data

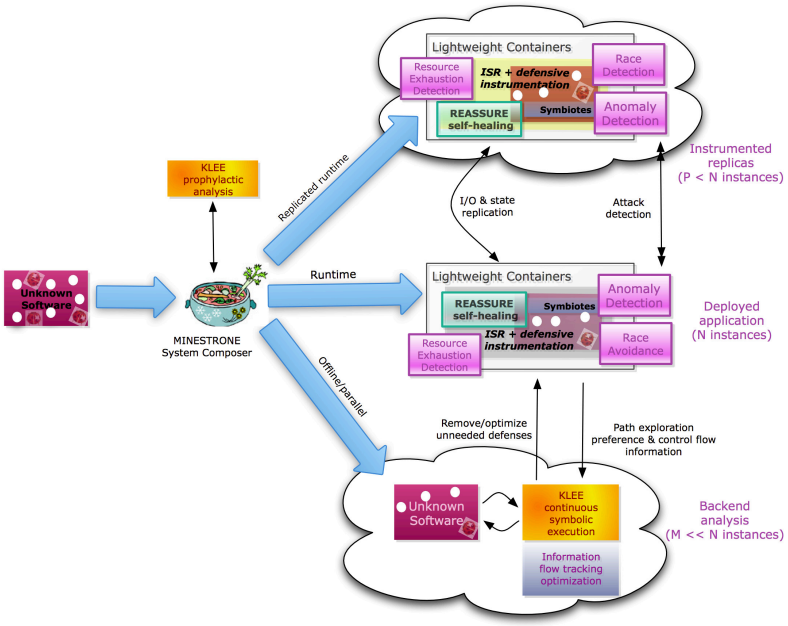

Fig. 1. **The MINESTRONE architecture and workflow. New software is processed by the MINESTRONE meta-compiler which, depending on configuration and environment capabilities, deploys several components. The production software (PS) is embedded within a lightweight OS-virtualization container which manages resource consumption and uses behavior anomaly detection. Inline reference monitors, in the form of binary- or source-rewriting are also embedded within the PS. A separate environment used for symbolic analysis is also created, with communication between PS and analysis. Diversified replicas (DRs) with similar instrumentation may also be set up, with I/O and state sharing performed with the PS. A given analysis instance may be used by one or more PS; likewise, each PS may be using one or more DRs.**

will allow us to focus further concurrent program analysis (through symbolic execution) to portions of the code that are more heavily exercised or are otherwise considered security-critical. We are using symbolic execution combined with static analysis to determine the safety properties of parts of software that is being deployed. Our symbolic execution framework [5], [6], [7] will explore possible execution paths of a program by analyzing it on unconstrained symbolic input and systematically following branches when the outcome depends on the symbolic input. Further, we will selectively integrate our previous static analysis and model checking frameworks [8], [9], [10], [11], [12], [13], [14], [15] into MINESTRONE to detect vulnerabilities such as number handling, error handling, concurrency handling, memory safety errors (*e.g.,* buffer over-flows/underflows), null pointer errors, and tainted data/input validation errors. On top of these program analysis frame-works, we plan to build several novel analysis techniques to improve the coverage of error detection, the soundness of confinement and diversification, and the speed of the entire system. While the initial static analysis may be done either at a centralized repository from which the software is downloaded or at the end-user machine, the symbolic execution component is likely to be centralized (*e.g.,* at the department or enterprise level) to avoid unnecessary duplication of significant effort.

Since symbolic execution of non-trivial software is a time-consuming endeavor, we proactively and comprehensively (except as otherwise indicated by static analysis) instrument software with code that detects and confines vulnerabilities. The nature of the instrumentation depends on the type of vulnerability. When source code is available, we will insert the instrumentation through source-code transformations [16], [17]. Otherwise, we will inject our instrumentation in program binaries using the PIN binary rewriting tool (which is neither as efficient nor as effective, due to the various challenges in working with binaries) [18], [4]. The specific vulnerability types we are protecting against include number handling, error handling, concurrency handling, memory safety errors (*e.g.,* buffer overflows/underflows), null pointer and tainted data/input validation errors.

Multithreaded code is difficult to write because developers must reason about all possible ways the threads may interact with each other. Due to the same complexity, multithreaded code is also difficult to debug and fix. For example, a study has shown that a significant number of concurrency error fixes did not fix the corresponding errors and, worse, introduced new errors [19]. This situation may worsen as developers are writing more multithreaded programs driven by the high performance demand and the current multicore trend. We propose to automatically avoid races[1] using a number of novel techniques. Mechanically, our approach will work as follows: (1) analyze applications to detect likely data races, (2) merge adjacent data races into atomic regions that match developer atomicity or ordering intents of code, and (3) defensively insert synchronization operations to prevent these likely races. The key advantage of this approach is to relieve developers from fixing many races, thus improving the reliability of multithreaded software.

[1]We use the term *race* to denote all non-deadlock concurrency errors, including low-level data races (*i.e.,* concurrent accesses to a shared variable with at least one write access), atomicity errors, and order errors [19].

One important and novel element of our approach involves protecting the inline reference monitor itself from silent (unobserved) compromise. We will achieve this using what we call "In-Code Execution" (ICE). ICE is inserted into the program code either using existing "memory gaps" between code and string structures or specially crafted padding generated during the analysis phase. ICE creates an independent execution context from the native program context at runtime making sure that all the necessary state information is preserved. The reference monitor executes as an encrypted payload spread out throughout the code. The ICE insertion and payload encryption engines can be modified at each run of the binary offering a diversification of the protection mechanism that elevates the protection of the reference monitor. It is important to note that ICE does not use traditional virtualization techniques only standard CPU instructions. We will also use lightweight OS-level virtualization to provide an additional layer of process confinement, including anomaly detection. This will let us interact with the OS and enforce resource usage limits, protect the inline reference monitor, and capture/inspect/duplicate I/O (especially in conjunction with diversified replicas).

Artificial code diversification (ACD), in the form of ASLR and Instruction Set Randomization (ISR) [20], [21], will act as one form of containment. We have extended our previous work on ISR to minimize its performance impact by reducing the portion of the program that must be run in the ISR runtime. This is done by concentrating ISR to the parts of the code that static analysis indicates is more likely to contain bugs and by randomizing selective portions of the program that cannot be avoided by a successful compromise (thus causing a program fault) [22]. We are developing a framework that allows the integration of any type of localized ACD toward detecting attacks. When local resources are available (*e.g.,* fast system with multicore CPU), our system executes multiple versions of the diversified code and compare results. If local resources are insufficient, our instrumentation will transmit all process I/O to a remote system that runs diversified replicas of the application. While this will only allow *ex post facto* intrusion detection, its relatively low performance impact on the protected system allows its use on low-power devices such as smartphones and netbooks. Remote execution of instrumented programs replicas also enables investigation and implementation of additional diversification and fault detection approaches that would not otherwise be possible with deployed program instances.

The feedback loop will enable our system to gradually and selectively remove instrumentation checks as symbolic execution indicates that specific parts of the code are not susceptible to certain vulnerabilities. Thus, over time the performance of a deployed piece of software will improve. Furthermore, the injected instrumentation will gather usage data that will allow us to concentrate the symbolic execution to the actively used portions of the program. Specific input vectors may also be supplied to the symbolic execution engine to facilitate its exploration of the code paths and state space.

As a specific example of such interaction, we briefly discuss how analysis can improve the performance of our dynamic confinement and self-healing components. As part of handling a security violation, these components may roll back a faulty execution to a series of recent checkpoints for recovery. Since taking a checkpoint is expensive, we will avoid doing so when possible by developing a purity analysis that analyzes the side effects of functions [23]. There are three cases we can skip checkpointing a function: (1) when the function is *pure*: it does not modify any memory location outside the stacks, allocate or deallocate any resource, or perform any I/O; (2) when the function is *on-error pure*: it does not have any side-effect when it returns an error; and (3) when the function is *partially pure*: it has side effects only on some of its execution paths, so we only need to checkpoint for those paths.

While general purity analysis has been previously studied [23], [24], we are the first to propose on-error purity and partial purity. Further, a key novel feature of our analysis is *return-code-sensitive*, *i.e.,* it will use the results from our error-code analysis to compute different side-effect summaries for success returns and error returns. Such return-code-sensitivity strikes a good balance between precision and scalability and is valuable for other kinds of static analysis as well, as functions tend to do very different things on success and error. We plan to extend return-code sensitivity to other types of static analysis as well.

In addition to designing, prototyping and evaluating the overall `MINESTRONE` architecture, we seek to advance the state of the art in each of the component areas, by expanding the scope of program analysis techniques to new vulnerability classes, developing self-protecting confinement mechanisms, and creating diversification schemes that offer highly tunable performance-security tradeoffs.

III. Current Status

Our primary task in realizing this research vision is the development of an integrated architecture that combines static and program analysis, confinement, and diversification in a feedback system that allows for continuous improvement of the security and performance of the protected software. Therefore, our subtasks relate to the development of individual mechanisms within each of these areas, and the integration into a single system. We have been working on this project vision since August 2010. In that time, we have refined the architecture and have been working toward building the individual components:

- We have developed a binary IRM that implements selective ISR, Write Integrity, self-healing, and taint tracking with advanced performance optimizations to remove unnecessary instrumentation. For example, in some applications the overhead of ISR is less than 1%, while we have reduced the overhead of taint analysis by 40% to 60% over the best reported implementation [25], [26].
- We have developed techniques for analyzing programs to identify and mitigate concurrency bugs [27], [28]. Using static analysis and schedule memoization, we can force safe thread

scheduling with modest performance impact.

• We have developed versions of ICE that can be embedded in binaries for different architectures (ARM, MIPS, x86). We have demonstrated ICE for such diverse environments as Cisco routers and Android handsets. The ICE implementation is very efficiently executed utilizing the raw computational resource of the hardware platform, bypassing layers of overhead produced by operating systems or VMs that host an OS. One advantage of an ICE security payload over a reference monitor is better performance.

• We have augmented our initial lightweight container scheme with full-process logging/replay and system call monitoring. This capability, combined with the self-healing component in our IRM, will allow us to do fast rollback.

• We have scaled up the symbolic execution component, with respect to the state space explored. Our new techniques allow us to identify equivalent states (and thus avoid them), yielding a 9-fold performance speedup on average.

• We are developing a I/O redirection prototype that covers interactive applications. We are integrating this with our diversification IRM and the lightweight containers, allowing us to place and move replicas in any number of systems and to deploy the detection and mitigation techniques we develop.

The main challenge with our work will be managing the integration complexity, conducting realistic experiments, and developing a management framework that makes it easy to use `MINESTRONE`. This will be the focus of our future efforts.

Acknowledgements*:* This work was supported by the US Air Force through Contract AFRL-FA8650-10-C-7024. Any opinions, findings, conclusions or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government or the Air Force.

REFERENCES

[1] Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient software-based fault isolation. In: In Proceedings of the 14th ACM Symposium on Operating Systems Principles. (1993) 203–216

[2] Seward, J., Nethercote, N.: Valgrind, an open-source memory debugger for x86-linux. (http://developer.kde.org/~sewardj/)

[3] Rinard, M.C., Cadar, C., Dumitran, D., Roy, D.M., Leu, T., Beebee, W.S.: Enhancing server availability and security through failure-oblivious computing. In: OSDI. (2004) 303–316

[4] Sidiroglou, S., Laadan, O., Viennot, N., Perez, C.R., Keromytis, A.D., Nieh, J.: ASSURE: Automatic Software Self-healing Using REscue points. In: Proceedings of the 14^{th} International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). (2009) 37–48

[5] Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8^{th} Symposium on Operating Systems Design and Implementation (OSDI). (2008) 209–224

[6] Yang, J., Sar, C., Twohey, P., Cadar, C., Engler, D.: Automatically generating malicious disks using symbolic execution. In: Proceedings of the 2006 IEEE Symposium on Security and Privacy (SP '06). (2006) 243–257

[7] Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06). (2006) 322–335

[8] Engler, D., Chelf, B., Chou, A., Hallem, S.: Checking system rules using system-specific, programmer-written compiler extensions. In: Proceedings of Operating Systems Design and Implementation (OSDI). (2000)

[9] Engler, D., Yu Chen, D., Hallem, S., Chou, A., Chelf, B.: Bugs as deviant behavior: A general approach to inferring errors in systems code. In: Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01). (2001)

[10] Engler, D., Ashcraft, K.: RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In: Proceedings of ACM SOSP. (2003)

[11] Yang, J., Kremenek, T., Xie, Y., Engler, D.: MECA: an extensible, expressive system and language for statically checking security properties. In: Proceedings of the 10^{th} ACM Conference on Computer and Communications Security (CCS). (2003)

[12] Ashcraft, K., Engler, D.: Using programmer-written compiler extensions to catch security holes. In: Proceedings of the 2002 IEEE Symposium on Security and Privacy (SP '02), Oakland, California (2002)

[13] Yang, J., Chen, T., Wu, M., Xu, Z., Liu, X., Lin, H., Yang, M., Long, F., Zhang, L., Zhou, L.: MODIST: Transparent model checking of unmodified distributed systems. In: Proceedings of the 6^{th} Symposium on Networked Systems Design and Implementation (NSDI). (2009)

[14] Yang, J., Sar, C., Engler, D.: Explode: a lightweight, general system for finding serious storage system errors. In: Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI '06). (2006) 131–146

[15] Yang, J., Twohey, P., Engler, D., Musuvathi, M.: Using model checking to find serious file system errors. In: Proceedings of the 6^{th} Symposium on Operating Systems Design and Implementation (OSDI '04). (2004) 273–288

[16] Sidiroglou, S., Keromytis, A.D.: Execution Transactions for Defending Against Software Failures. International Journal of Information Security (IJIS) **5** (2006) 77–91

[17] Sidiroglou, S., Locasto, M.E., Boyd, S.W., Keromytis, A.D.: Building A Reactive Immune System for Software Services. In: Proceedings of the 11^{th} USENIX Annual Technical Conference. (2005) 149–161

[18] Kim, H.C., Keromytis, A.D.: On the Deployment of Dynamic Taint Analysis for Application Communities. IEICE Transactions **E92-D** (2009) 548–551

[19] Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems, New York, NY, USA, ACM (2008) 329–339

[20] Kc, G.S., Keromytis, A.D., Prevelakis, V.: Countering Code-Injection Attacks With Instruction-Set Randomization. In: Proceedings of the 10^{th} ACM Conference on Computer and Communications Security (CCS). (2003) 272–280

[21] Boyd, S.W., Keromytis, A.D.: SQLrand: Preventing SQL Injection Attacks. In: Proceedings of the 2^{nd} Applied Cryptography and Network Security Conference (ACNS). (2004) 292–302

[22] Locasto, M., Wang, K., Keromytis, A., Stolfo, S.: FLIPS: Hybrid Adaptive Intrusion Prevention. In: Proceedings of the 8^{th} Symposium on Recent Advances in Intrusion Detection (RAID). (2005) 82–101

[23] Landi, W., Ryder, B.G., Zhang, S.: Interprocedural side effect analysis with pointer aliasing. In: PLDI '93: Proceedings of the 1993 ACM SIGPLAN conference on Programming language design and implementation. (1993)

[24] Xu, H., Pickett, C.J.F., Verbrugge, C.: Dynamic purity analysis for java programs. In: PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. (2007)

[25] Portokalidis, G., Keromytis, A.D.: Fast and Practical Instruction-Set Randomization for Commodity Systems. In: Proceedings of ACSAC. (2010)

[26] O'Sullivan, P., Anand, K., Kothan, A., Smithon, M., Barua, R., Keromytis, A.D.: Retrofitting Security in COTS Software with Binary Rewriting. In: Proceedings of the 26^{th} IFIP International Information Security Conference (SEC). (2011)

[27] Cui, H., Wu, J., che Tsai, C., Yang, J.: Stable Deterministic Multithreading through Schedule Memoization. In: Proceedings of the 9^{th} Symposium on Operating Systems Design and Implementation (OSDI). (2010)

[28] Wu, J., Cui, H., Yang, J.: Bypassing Races in Live Applications with Execution Filters. In: Proceedings of the 9^{th} Symposium on Operating Systems Design and Implementation (OSDI). (2010)