Repeatable Reverse Engineering with the Platform for Architecture-Neutral Dynamic Analysis

Ryan J. Whelan, Timothy R. Leek, Joshua E. Hodosh, Patrick A. Hulin, and Brendan Dolan-Gavitt

Many problems brought on by faulty or malicious software code can be diagnosed through a reverse engineering technique known as dynamic analysis, in which analysts study software as it executes. Researchers at Lincoln Laboratory developed the Platform for Architecture-Neutral Dynamic Analysis to facilitate analyses that lead to profound insight into how software behaves. This tool was recognized with a 2015 R&D 100 Award for being one of the year's 100 most innovative technologies.

» **Billions of lines of computer code direct** the flow of information that drives the world's activities. This vast amount of code powers software programs that instruct systems to perform tasks as commonplace as word processing and as specialized as analyzing DNA sequence data. However, lurking within this benign software are critical vulnerabilities that cyber criminals exploit to steal or corrupt information. In addition, as new software versions, capabilities, and operating systems are introduced to the marketplace, older software code often becomes incompatible with new technology, rendering the software either ineffective or completely unusable. Although the U.S. government and businesses annually spend millions of dollars to recover from attacks that inject malicious software, or malware, into their computer systems and to keep their software operational, more effective analysis capabilities are still needed to enable rapid, successful diagnosis and resolution of software problems. Lincoln Laboratory researchers have created an open-source tool, the Platform for Architecture-Neutral Dynamic Analysis (PANDA), for analysts to use to quickly develop instrumentation that helps answer complex questions about software and that informs appropriate responses to malware intrusions.

Reverse Engineering

PANDA facilitates an analysis technique known as reverse engineering (RE), i.e., the process of analyzing a program's code to discover its undocumented internal principles. By closely inspecting the binary code that runs a piece of software, an analyst can study how the program has been constructed to perform its operations. Reverse engineering is frequently employed to enable legacy code to continue functioning, to identify vulnerabilities in software, and to understand the true purpose and actions of a software program.

It is common for legacy code to stop working as the software ecosystem surrounding it evolves. When that failure happens, and when corporate support for the old program has also long terminated, RE is the most cost-effective avenue to revive the functionality of the software. Using RE, analysts can discover the inputs and outputs to, and the dependencies and requirements of, a software program so that they can then develop appropriate fixes that allow the old code to run in a more modern environment.

Accurately identifying vulnerabilities is usually impossible without detailed RE knowledge. Analysts might be able to observe that a software bug exists, but being able to determine if it is exploitable, and therefore a critical vulnerability, is a much more difficult problem. Part of the solution to this problem is the determination of which specific parts of the program are questionable; often, source code is not available to help make this determination. Thus, without either performing RE or making use of the RE efforts of others, it is difficult to discriminate between unimportant bugs and serious vulnerabilities.

Vetting software to determine if it does what it is purported to do and nothing else is an important, complicated task. When the code is believed to be malware, this determination is usually obvious. However, we believe there is an increasingly fine distinction between malware and misbehaving code. Consider a program written by a legitimate, large U.S. company, and imagine that its code performs a host of unintended malicious actions, such as accessing personal information or modifying system settings. None of this behavior is indicated in the documentation or advertising literature, nor is it clearly essential for the primary purpose of the software. How is this code functionally distinct from malware? This scenario is not simply a thought experiment: in 2005, Mark Russinovich, the cofounder of Winternals Software, discovered that audio CDs produced by Sony BMG Music Entertainment were installing a rootkit onto millions of computers [1]. The Sony rootkit recorded information about users' computers to send back to Sony and hid every file on users' systems with a certain prefix; worse, Sony's uninstaller allowed any webpage to download and execute arbitrary code [2].

Reverse Engineering Through Dynamic Analysis

One approach to RE is *static analysis*. In this approach, analysts use tools such as disassemblers and decompilers to translate binary code into a form more easily read. Humans painstakingly navigate these representations, adding extensive annotations to ultimately reassemble a picture of how code and data operate at various levels of abstraction. *Dynamic analysis* is another approach to RE. In a dynamic analysis, while software executes on the system, analysts observe its behavior.

PANDA is fundamentally a dynamic analysis tool that can help analysts gain deep insight into software code by observing the code's behaviors across all levels of the operating system. Figure 1 provides a high-level overview of PANDA, and its use is depicted in Figure 2. First, an analyst captures a recording of some whole-system execution that he or she wishes to understand thoroughly. Then, the analyst writes analysis code in the form of plugins, which are modules that add specific capabilities to the software. Plugins collect data and consult or control other plugins. They are typically written quickly and iteratively, running a replay of the previously gathered recording over and over to construct a deeper understanding of the important aspects of system execution. For example, an initial plugin might just get a rough outline of what processes execute on the system and when key operating system events happen during the replay. A second analysis pass over the replay might focus in on the activity of a particular program or a portion of the replay. Further iterations over the replay might be more complex and allow analysts to selectively label interesting data and track those data as they flow around the system; this process is metaphorically similar to a positron emission tomography (PET) scan [3], which provides diagnostic scans of organs and tissues by tracing a radioactive substance as it travels through the body. We have found that this workflow powerfully enhances RE, as it enables analysts to iteratively build knowledge about dynamic software executions.

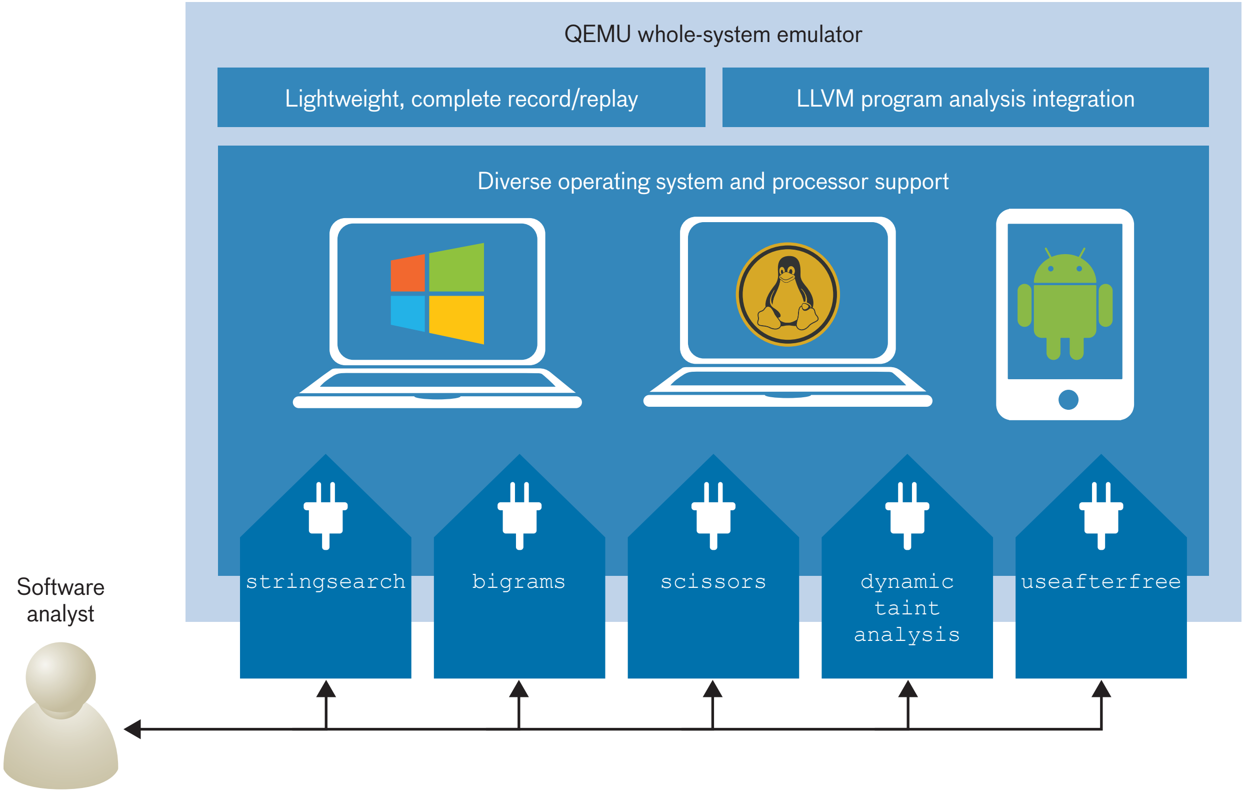

QEMU whole-system emulator
Lightweight, complete record/replay
LLVM program analysis integration
Diverse operating system and processor support
Software analyst
stringsearch
bigrams
scissors
dynamic taint analysis
useafterfree

FIGURE 1. This high-level overview of PANDA shows its key features. PANDA has the ability to efficiently record and replay whole-system executions; the ability to support diverse operating systems, such as Windows, Linux, and Android; and a modular software design in which each analysis can be implemented as a plugin and the plugins can be used in conjunction with one another. Plugins can execute a number of diverse tasks according to how they are programmed by the analyst. For example, they can track information about which processes are executing, enable dynamic searching of data in the system, perform automated web-traffic decryption for certain algorithms, and perform detailed exploit analysis.

PANDA System

PANDA is largely based upon the open-source whole-system emulator known as QEMU (Quick Emulator). QEMU is a robust platform that uses binary translation to support multiple processor architectures. Utilizing QEMU allows us to emulate an entire Windows or Linux desktop, an Android phone, and other embedded systems.

PANDA has four key features: the ability to record and replay entire software executions, an extensible plugin architecture, the ability to extend software analyses across multiple processor architectures, and the ability to emulate Android systems.

Record and Replay

PANDA's record and replay feature is conceptually simple. At the beginning of recording, we take a snapshot of the machine state, which includes the contents of registers and memory. Then, we record to a log all sources of nondeterministic data entering the system, which primarily includes the sources of input and output, such as network traffic and hard-drive data, but also includes other low-level sources that we have identified in the system. When any of these inputs comes into the system, we also record the information needed for us to determine when to replay the input.

PANDA's replay function, which has been tested extensively on two processor architectures (32- and 64-bit x86 and ARM), is quite stable and effective. It can record boot for a variety of operating systems; this action is challenging because of the complexity of the boot operation. PANDA recordings are also fairly compact in size even though our record log must capture the contents of all inputs into the system. Table 1 gives the record log sizes for a number of workflows. The modest size of these files makes them ideal for sharing and thus for enabling repeatable experiments. One of this article's authors has set up a website from which any of these and a number of other replay files can be downloaded and analyzed independently.[1]

Because PANDA allows full repeatability of replays, it is incredibly useful for dynamic analysis. Traditionally, manual dynamic analysis involves running a program inside a debugger and using the debugger to periodically inspect the program state. However, debuggers largely cannot execute backwards, so in order to inspect an earlier program state, an analyst must restart the program from scratch. This restart not only adds time to the analysis process but also changes many dynamic aspects of the program. With PANDA replays, dynamic information is the same each time, so information about the state of memory can be built up piece by piece, greatly accelerating RE.

Plugin Architecture

PANDA plugins take the form of shared libraries that can be loaded at any time during an analysis. The plugins are event-driven; that is, they perform tasks in response to events in the system that are specified by analysts' instructions. The analysts perform system instrumentation by using interfaces that have been made available in PANDA.

Many plugins depend on some common functionality. To avoid duplicating functionality throughout plugins while keeping the core of PANDA simple, we have implemented a mechanism for plugin-plugin interaction to allow individual plugins to expose a public interface that other plugins can utilize. The plugin-plugin interaction allows code reuse and reduces the duplication of specialized code that is used for complex analyses.

Architecture-Neutral Analysis

A number of dynamic analyses that happen at the system instruction level are invaluable for RE. For instance, in taint analysis, data in the system are labeled (tainted) and then tracked to enable a detailed understanding of the true information-flow patterns around, in, and out of a system. This analysis can be thought of as a PET scan for a computer [3]. In order to properly track labels, one must perform an additional complex analysis alongside every system instruction. Some of the complexity of these additional analyses is due to the differences in processor architectures of systems. For example, desktop architectures (such as x86) are more complex than power-constrained architectures (such as ARM).

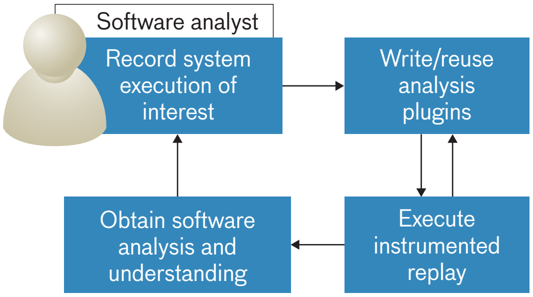

Software analyst
Record system execution of interest
Write/reuse analysis plugins
Obtain software analysis and understanding
Execute instrumented replay

FIGURE 2. In the replay-based reverse engineering (RE) workflow, PANDA can record and replay whole-system executions. This capability is the foundation of PANDA's use in RE. To use PANDA, the analyst captures a recording and then iteratively uses or builds data analyses to incrementally build RE knowledge.

Table 1. Record Log for Various Replays

REPLAY	INSTRUCTIONS (BILLIONS)	LOG SIZE (MB)
Operating system boot	9.3	533.0
Spotify playing a song snippet	12.0	229.0
Malware recording	9.1	43.0
User browsing to a website	8.6	9.4

PANDA avoids the difficulties associated with supporting multiple processor architectures by performing analyses in a generic intermediate representation that is not specific to a particular processor architecture. We perform dynamic binary translation, which is the process of translating the code under analysis to the intermediate representation, to enable the generic analyses. Dynamic binary translation is the underlying technology that makes some of our novel analyses possible.

[1] http://www.rrshare.org

Android Support

An emulator similar to PANDA is included in Google's Android software development kit and contains the necessary emulated hardware to produce a realistic Android environment. In order to provide Android support to PANDA, we ported the features necessary to emulate devices that are unique to Android phones. Significant additional work was required to fully support modern Android emulation: integrating telephony, camera, and Android debug bridge support; integrating secure digital (SD) memory card support; translating inputs for graphical interfaces; supporting common formats for the storage devices; arranging to support PANDA's record and replay mechanism; and employing various other bug fixes, including one for a graphics bug.

Plugin Details

To date, more than 40 robust analysis plugins that can be applied to RE have been developed for PANDA by Lincoln Laboratory researchers, collaborators at a number of universities, and the open-source community at large. These plugins are available in our github repository at https://github.com/moyix/panda. The following novel plugins have proven particularly useful in RE.

Tappan Zee (North) Bridge

Reverse engineering tasks often hinge on finding out what piece of code either implements some high-level functionality or handles some particular data. In large programs, these discoveries can be quite difficult. When the data are a fixed string embedded in the program, analysts are usually able to easily determine the function of a piece of data, but when the data are dynamic, analysts must laboriously trace the flow of data from some known input source through a chain of intermediate functions to the location where it is finally used. Moreover, when the data sought are some intermediate values not directly derived from the input, even this approach may fail.

In previous work [4], we developed a system, Tappan Zee (North) Bridge (TZB), for locating points at which we can interpose on memory accesses in a system to monitor events during system execution. We have since discovered that TZB is also immensely useful for RE. The central concept behind TZB is that memory accesses can illuminate the internal details of a system. As a program runs, functions called from different contexts read and write input, output, and intermediate results to memory. By appropriately separating out these memory accesses according to program and calling context (Figure 3), we obtain coherent streams of data that can then be searched and analyzed for information of interest. These streams of data accessed at a particular point in a program are called tap points because they are places in the code where one might "tap" to get useful information from a system.

In the simplest case, we may wish to find out what part of a program handles a certain bit of data, such as a string we type into a program, or what function causes a particular string to be printed to the console or shown in the user interface. For such tasks, searching all tap points for some fixed strings will give us a set of functions that were seen to read or write data matching our search string. To accomplish this type of search, we created the `stringsearch` plugin, which tracks all memory accesses made in the system; splits them up according to the calling context, program counter, and address space; and then searches the resulting streams for a list of keywords.

Sometimes, we may not know the exact format of the data, but we may know some statistical features of it. For example, if we are searching for a digital rights management (DRM) decryption function, we know from previous work by Wang et al. [5] that the inputs to such functions have high byte entropy and are statistically random (according to a test such as Pearson's chi-squared test), but their outputs are not random. We recently used Pearson's test to locate the DRM decryption function within Spotify and showed that this function could be used to extract unencrypted audio files [6].

To support this latter type of search, PANDA can use appropriately named `unigrams` and `bigrams` plugins to collect unigram and bigram (one- or two-unit sequences in a string) statistics about each tap point. These functions collect unigram and bigram histograms for data read or written at each tap point during an execution. Once these histograms are gathered, an analyst can write scripts to compute statistical features, such as byte entropy, chi-squared values, or a distance measure (such as Kullback–Leibler divergence) to a previously observed distribution.

The ability to search through tap points for data of interest can allow a reverse engineer to quickly zero in on the parts of a program of greatest interest or to extract normally unobservable data from a program as it runs.

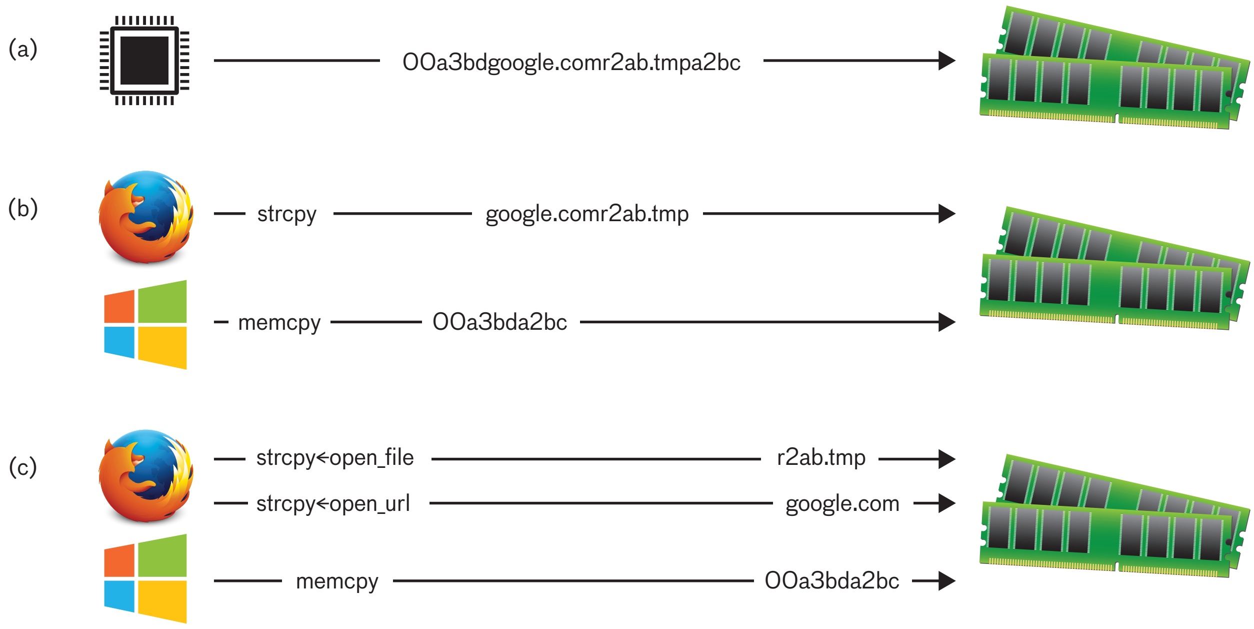


FIGURE 3. Memory accesses are made by a program with varying amounts of context: (a) presented as a single stream of information from the CPU to memory, (b) split up according to program, and (c) split up according to program, location within the program, and calling context.

The TZB system depends crucially on PANDA's record and replay functionality: on a live execution, the amount of compute resources needed to identify and halt on every memory access and inspect its contents would cause the operating system to become prohibitively slow and impossible to analyze. In a replayed execution under PANDA, plugins can have arbitrary performance overhead without perturbing the events in the replay.

Scissors

One of PANDA's biggest contributions to RE is its ability to do offline analysis, that is, to collect a recording of a system's execution at normal speed and then replay that execution with heavyweight analyses running, potentially over a long period of time. Still, many analyses are too computationally intensive to be tractable over replays that have potentially billions of instructions. To address this issue, we created the `scissors` plugin, which enables the user to excise smaller portions of replays and then analyze just the shortened portion. Combined with the ability of components such as TZB to rapidly locate sections of interest in the replay, the `scissors` plugin allows analysts to focus their attention on key events during execution and ignore everything else.

Taint Analysis

As previously mentioned, taint analysis is the process of tagging data in the system with labels and then tracking those labels to enable a detailed understanding of the true information-flow patterns around, into, and out of a system. We have implemented dynamic taint analysis as a PANDA plugin that permits precise labeling of data in a number of ways, such as labeling file contents, network data, raw memory, and CPU registers. These labels are then tracked automatically and stored in a shadow memory that associates tainted memory, registers, and input/output buffers with label sets.

Unlike other existing taint analysis systems, our system is independent of the underlying processor architecture and has been used to analyze replays based on the x86, x86-64, and ARM processor architectures. We can also easily extend our taint analysis system to all of the architectures that our system emulator supports.

Our system includes query mechanisms that allow an analyst to ask if data are tainted at some replay point and to examine the set of associated taint labels. These mechanisms permit the analyst to ask very detailed questions about the software they are analyzing: If I mark incoming network traffic as tainted, where does it flow through the system? Is any of this traffic interacting with vulnerable code that could potentially be exploited by an adversary? If I mark sensitive files as tainted, are they ever unknowingly exfiltrated? How much control does a potential adversary have over untrusted data at various points in the execution of a program?

Many of the specifics of how this subsystem was designed and implemented have been described in our previous work [7]. Here we describe some salient features of PANDA that enable analysts to observe software in detail:

- Whole-system support—PANDA's taint analysis tracks labels even if they flow between different processes and privilege levels.
- Input/output support—Because our shadow memory includes the hard drive, network card, and associated input/output buffers, an analyst can precisely introspect into how data propagate through the system at a low level and can properly track how data move through these devices.
- Replay-based taint analysis—Our taint analysis plugin uses a record and replay system to turn an intractable online analysis into a tractable offline analysis. For many platforms, such as Android, even pure emulator-based execution is barely fast enough to prevent operating system and networking timeouts during taint analyses, which are typically computationally intensive.
- Detail and fidelity—Taint analysis in PANDA focuses primarily on the detail that can be obtained from the analysis. For instance, a file can be labeled such that every byte in the file gets a different label. Further, computation is modeled with high fidelity by tracking detailed metadata with each byte of memory, allowing an analyst to measure how much the tainted data have changed since they were originally labeled.
- Interface—Taint labeling and querying can be either driven by events (through callbacks registered with the taint plugin) or invoked by a call to the interface exposed by the plugin. This choice of approaches provides flexibility to the analyst.

Case Studies

The following three RE use cases for PANDA illustrate the system's capabilities. In the first example, we revived an old version of the game StarCraft for which the CD key had been lost; with PANDA's plugins, we were able to rapidly locate the key verification code and harness it to produce keys on demand. In the second, PANDA's whole-system replay function enabled us to perform an in-depth diagnosis of a Windows Internet Explorer vulnerability to characterize this vulnerability as a use-after-free bug (i.e., an attempt to access previously deallocated memory). In the third, an Android chat client suspected of censoring messages was quickly determined to be doing so via a censorship blacklist that was readily extracted. Note that, while we used some of the plugins that were mentioned earlier in this article, we did not apply all of them to our use cases; rather, we allowed the task at hand to drive the choice of plugins to employ.

Reviving Legacy Code

StarCraft is a science fiction video game released in 1998 by Blizzard Entertainment. Each of the game's discs comes with a unique CD key that identifies the copy and permits both installation and online play. Originally, CD keys were 13 numbers, but Blizzard revised later copies of the game to use keys consisting of 26 alphanumeric characters. The original 13-number format was very simple to reverse engineer; however, if you legally purchased a newer copy of the game and lost your 26-character CD key, you would be unable to install and play the game.

We used PANDA to find and rapidly reverse engineer the 26-character CD key validation algorithm for StarCraft. First, we collected a recording of the StarCraft installer rejecting a random sequence of letters and numbers. We then provided both this incorrect key sequence and the text of the rejection dialog as searches to PANDA's TZB, which promptly found both in the replay. This discovery focused our attention on about 200,000 instructions out of the 60 million in the complete replay (a 300-fold reduction). We then used the `scissors` plugin to extract just this operative segment containing the validation algorithm.

Through manual static analysis of the code in the remaining replay segment, we ascertained that the installer decrypts the CD key and checks the high-order bits of the resulting 120-bit integer against a fixed value. This "magic number" is not immediately apparent in the code's disassembly, but a simple PANDA plugin was rapidly written that printed the magic number out when it was read from memory through the use of a concept similar to the `stringsearch` plugin. Our analysis showed that the fixed value was 23. Manual RE from there easily revealed the complete key-computation algorithm. Some additional mathematical analysis indicated a very low key density: only 1 in 27,000 of the possible CD keys are actually valid.

We then used the Hex-Rays decompiler to recreate source code of interest identified by the `stringsearch` plugin. The extracted code was harnessed as a decoder in a small program that was fed random keys to determine which ones were valid. Overall, this RE effort was very successful. PANDA allowed us to reduce the replay to a size at which a complicated analysis was immediately tractable with the `scissors` plugin, rapidly locate the code of interest for key validation with the `stringsearch` plugin, and ultimately to play our StarCraft game again by using a validated CD key generated by our extracted test harness.

Deep Vulnerability Diagnosis

Software vulnerabilities often have deep causes, with the underlying bug occurring well before a potential crash or exploit. One classic example is the use-after-free bug that exploits a program's retention of information referencing invalid, deallocated memory. When a program accesses this invalid memory, the program may crash because its data structures have been corrupted; however, the crash itself will give no hint about its underlying cause—e.g., where the bug was created, when the memory was freed, or even if the bug involved a use-after-free exploit.

As an experiment to test the effectiveness of PANDA in finding deep vulnerabilities, we had a team member prepare a replay containing a known triggered vulnerability. This replay was then given out with no information other than the fact of an application's crash and the standard Windows error message, "Application has stopped working." First, we used the `replaymovie` plugin to make a series of captures from the screen and to then stitch them together into a video of replay execution. This video indicated that the failing process was Internet Explorer and that the vulnerability was triggered by loading a malicious website. We then used TZB to search for "<HTML" and "has stopped working"; this search gave us temporal bounds in the replay for the bug's location. The `scissors` plugin enabled us to reduce the size of the replay and conduct more heavyweight analyses. Using TZB again, we extracted all further output at the <HTML tap point, which was exactly the full webpage that triggered the bug. The webpage indicated that the vulnerability was probably a use-after-free bug.

We then wrote a custom PANDA plugin called `useafterfree` to detect use-after-free memory corruption situations. This plugin was written for a Windows operating system, but it could easily be adapted for other systems. It tracks calls to Windows' low-level memory allocation functions, and it maintains shadow lists of valid and invalid memory. When a pointer to invalid memory is used, a use-after-free has occurred and the plugin detects it.

In this case study that highlights the iterative approach analysts often take while performing RE tasks with PANDA, the repeatability of PANDA replays was a key advantage. Writing custom plugins to target a specific replay is easier than writing plugins that generalize over a broad set of situations.

Uncovering Censorship Blacklists

We cannot always trust that the software we use is acting in our interests. For example, it is not uncommon for instant messaging clients to actively censor the conversations of their users [8, 9]. Such censorship can either be performed on server-side operations or be accomplished by a client-side blacklist that is periodically updated. In the former case, PANDA can be of no help because there is no code available to run and examine in vivo; however, in the latter, PANDA can extract a list of censored words from the client. To test PANDA's ability to uncover such a list, we examined the free LINE messenger client for Android. Analysts from the Citizen Lab at the University of Toronto's Munk School of Global Affairs had previously investigated LINE to determine that it censors certain users [10].

For our analysis, we created a recording in which we launched the LINE messenger and sent an instant message to another user. The sent message did not include any content we thought might be censored. Simply by sending the instant message, we supposed that LINE would still have to load its list of censored words and check our message against it, thus leaving the list open to extraction by PANDA.

To find the encrypted wordlist, we employed the TZB plugin, supplying some guesses as to words that might be subject to censorship and then searching all memory reads and writes made by LINE for these words. This process gave us a set of tap points that contained the sensitive words. As we suspected, the words we sought were indeed included in LINE's list of censored words. By the end of our analysis, we discovered 536 specific words that LINE was censoring in a completely automated fashion. PANDA expedited our LINE analysis far beyond the level of time and effort we would have expended had we used a more manual approach.

Future Directions

We have been actively using PANDA for the past three years to quickly reverse engineer large, real-world software systems without the availability of source code. In most case studies during this time, we have found PANDA to be invaluable for speeding up RE, either by entirely obviating the need for manual analysis or by precisely directing human attention to the critical portions of a large code base.

In the near term, we are planning to enhance several key aspects of PANDA: performance, architecture support, and analysis capabilities. Performance can potentially be improved in our recording infrastructure and in many of our plugin implementations. PANDA is processor architecture–neutral in principle, but a number of features have not yet been ported to all supported processor architectures. For deep operating system analysis capabilities, we plan to create new plugins that encapsulate the domain-specific knowledge necessary to retrieve useful information about various operating systems.

In the long term, we wish to make this a tool that can be used by anyone to reverse engineer complex systems. We have released PANDA as an open-source tool to the cyber security community, and we have transitioned PANDA technology to several other programs within Lincoln Laboratory's Cyber System Assessments Group and to their respective sponsors within the Department of Defense. We hope that continued development by the open-source community and technical staff at Lincoln Laboratory will make this a common tool for dynamic software analysis and RE. ■

References

1. M. Russinovich, "Sony, Rootkits and Digital Rights Management Gone Too Far," Microsoft Server & Tools Blogs, 31 Oct. 2005, available at http://blogs.technet.com/b/markrussinovich/archive/2005/10/31/sony-rootkits-and-digital-rights-management-gone-too-far.aspx.
2. J.A. Halderman and E. Felten, "Sony's Web-Based Uninstaller Opens a Big Security Hole; Sony to Recall Discs," Freedom to Tinker blog, 15 Nov. 2005, available at https://freedom-to-tinker.com/blog/felten/sonys-web-based- uninstaller-opens-big-security-hole-sony-recall-discs/.
3. S. Mysore, B. Mazloom, B. Agrawal, and T. Sherwood, "Understanding and Visualizing Full Systems with Data Flow Tomography," *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008, pp. 211–221.
4. B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee, "Tappan Zee (North) Bridge: Mining Memory Accesses for Introspection," *Proceedings of the 2013 Association for Computing Machinery Special Interest Group on Security, Audit and Control, Conference on Computer and Communications Security*, 2013, pp. 839–850.
5. R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Steal This Movie: Automatically Bypassing DRM Protection in Streaming Media Services," *Proceedings of the 22nd USENIX Conference on Security,* 2013, pp. 687–702.
6. B. Dolan-Gavitt, "Breaking Spotify DRM with PANDA," Push the Red Button blog, 3 July 2014, available at http://moyix.blogspot.com/2014/07/breaking-spotify-drm-with-panda.html.
7. R. Whelan, T. Leek, and D. Kaeli, "Architecture-Independent Dynamic Information Flow Tracking," *Proceedings of the 22nd International Conference on Compiler Construction*, 2013, pp. 144–163.
8. J. Knockel, J.R. Crandall, and J. Saia, "Three Researchers, Five Conjectures: An Empirical Analysis of TOM-Skype Censorship and Surveillance," *Proceedings of IEEE Symposium on Foundations of Computational Intelligence*, 2011.
9. A. Senft, A. Sinpeng, A. Hilts, et al., "Asia Chats: Analyzing Information Controls and Privacy in Asian Messaging Applications," The Citizen Lab's Reports and Briefings, 14 Nov. 2013, available at https://citizenlab.org/2013/11/asia-chats-analyzing-information-controls-privacy-asian-messaging-applications/.
10. S. Hardy, "Asia Chats: Investigating Regionally-Based Keyword Censorship in LINE," The Citizen Lab's Reports and Briefings, 19 Nov. 2013, available at https://citizenlab.org/2013/11/asia-chats-investigating-regionally-based-keyword-censorship-line/.

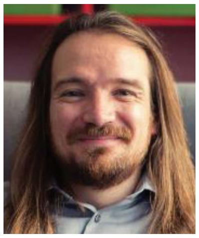
About the Authors

Ryan J. Whelan is a member of the technical staff in the Cyber System Assessments Group at Lincoln Laboratory. He currently conducts research in the areas of static and dynamic program analysis and transformation, compiler design and implementation, malware analysis, software reverse engineering, vulnerability discovery, and instrumentation techniques for cyber security. Prior to joining the Laboratory, he earned bachelor's and master's degrees in computer engineering from Northeastern University. During that time, he completed several cyber security internships at Lincoln Laboratory.

Timothy R. Leek is a member of the technical staff in the Cyber System Assessments Group. He is currently researching automated techniques for finding errors in programs by using a combination of static and dynamic analysis and some machine learning. He joined Lincoln Laboratory in 2001 as a member of the Ballistic Missile Defense System Integration Group, where he investigated machine learning techniques for building decision architectures. Prior to joining the Laboratory, he worked as a technical writer for U.S. Surgical Corporation, as a scientific programmer at Yale University, and as a systems administrator at the Salk Institute for Biological Studies in La Jolla, California. After earning a master's degree in computer science at the University of California, San Diego, he joined Bolt Beranek and Newman, where he applied statistical methods to information retrieval, topic detection and tracking, machine translation, and summarization. He also holds bachelor's degrees in physics and English from the University of Connecticut.

Joshua E. Hodosh is a member of the technical staff in the Cyber System Assessments Group and has been at Lincoln Laboratory since 2010. He holds a bachelor's degree in computer science from Rensselaer Polytechnic Institute and a master's degree in computer science from Northeastern University.

Patrick A. Hulin joined the Cyber System Assessments Group in 2014. His current research lies primarily in the areas of virtual machine introspection, automated software reverse engineering, dynamic program analysis, and vulnerability understanding. He holds a bachelor's degree in mathematics from MIT.



Brendan Dolan-Gavitt is an assistant professor of computer science at the New York University Polytechnic School of Engineering. Previously, he worked as a postdoctoral researcher in the Intrusion Detection Systems Lab at Columbia University. His research interests lie in the area of systems security, particularly in the development of automated techniques for understanding computing systems and the application of that understanding to the creation of novel defenses. He is also active in the rapidly growing field of memory forensics and has published papers on extracting forensically relevant information from images of RAM. He received his doctoral degree from the Georgia Institute of Technology and his bachelor's degree in mathematics and computer science from Wesleyan University. He also spent two years working as an information security analyst and researcher for the MITRE Corporation.