Band-aid Patching*

Stelios Sidiroglou
stelios@cs.columbia.edu
Columbia University

Sotiris Ioannidis
si@cs.stevens.edu
Stevens Institute of Technology

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

Abstract

Testing vendor-issued patches remains one of the major hurdles to their speedy deployment. Studies have shown that administrators remain reluctant to quickly patch their systems, even when they have the capability to do so, partly because security patches in particular are often incomplete or altogether non-functional.

We propose Band-aid Patching, a new approach for concurrently testing application patches. Using binary runtime injection techniques, we patch binaries such that when program execution reaches a program segment that has been affected by an issued patch, two (or more) program execution threads are created. These threads speculatively execute both parts of the code (patched and unpatched). Our system then retroactively selects one of the execution threads based on a variety of criteria, including obvious faultiness, prior history, and user input. We believe this approach to offer significant advantages to accelerating deployment of hot fixes while providing some assurance to system administrators. In this paper, we describe our initial thoughts on the system architecture, and provide some preliminary indications on the feasibility and performance impact of our scheme.

1 Introduction

Despite considerable efforts in software reliability, software bugs account for more than 40% of system failures [8]. Even more disheartening is that retroactively dealing with bugs, *e.g.,* with software patches, reveals latent bugs 70% of the time [4, 12] leaving system administrators stuck between a rock and a hard place .

Given that many system bugs often lead to system vulnerabilities, system administrators need to perform the highly dextrous task of juggling between system down-time and installing updates. Unfortunately, in many cases the cost or complexity of launching another machine to test the effects of patches is non-trivial.

Currently, the safest option available to system administrators, even with dynamic software updating [2, 10], is to test patches on non-production machines and mirror traffic to the application. Since the machines are disjoint, comparison of two or more machines with mirroring requires some level of cross-system synchronization, adding considerable overhead. Problems arise with the use of encryption and protocol non-determinism in the mirrored systems. What we need is the ability to test patches on running systems with impunity and little cost.

With these constraints in mind, we present *Band-aid Patching,* a new approach for testing application patches prior to definitive deployment. Using binary runtime injection techniques, we modify binaries so that when program execution reaches a program segment that has been affected (modified) by an issued patch, the program "forks" speculative execution threads that cannot interfere with each other. The output of these threads is examined to discover misbehavior. Our system contains the effects of any misbehavior by unrolling their execution results, and selecting from among the non-misbehaving execution threads. Often, identifying the misbehaving thread is easy: based on its actions (*e.g.,* causing a program crash). Such problems may occur in either the original or the updated version of the code (or, if we are unlucky, in both).

The main advantage of our approach lies in its ability to allow for simple dynamic patches to be tested side-by-side with production-level application state. The output created by these "patch threads" can be then compared, to provide some guarantee over the correctness of the computation. Since there is a cost associated with our approach, our scheme works best when the patches are relatively small in size, or when they affect code that is not in the critical path of the application. Generally, security and stability patches fall in this category. Although our scheme could be applied against any type of patch (*e.g.,* large patches introducing new functionality), the cost and complexity associated would likely be too high.

As a preliminary validatation of our approach, we developed a proof-of-concept implementation of Band-aid

Patching using source-to-source transformations and the *dyninst* [3] runtime instrumentation tool. Specifically, we use dyninst to insert instrumentation trampolines that point to the different versions of the patches under test, and instrumentation that examines and compares the output of these patches. The type of patches that can be applied can range from simple logic bug fixes to testing new vulnerability protection mechanisms. For more complex updates, *i.e.,* where changes to function prototypes and type definitions are required, our approach could be combined with more sophisticated dynamic software update techniques [5, 10]. Other promising recent work in this space [6] uses virtual machines combined with vulnerability-specific predicates that are executed in the context of the vulnerable process to test whether a flaw is being exercised (or was exercised at some point in the past, if such logs are available). However, these predicates must be generated manually, and require intimate knowledge of the application and the patch (or the vulnerability). In contrast, our technique uses only the vendor-issued patch and can be applied with little or no intervention by users or administrators. In this preliminary work, we expose some of the issues associated with our approach and identify areas for future investigation.

2 Approach

Conceptually, our approach can be described by two components: an execution module and a decision module.

The execution module provides the speculative execution environment where patches can be applied to service instances. It is possible that multiple such instances run in parallel while being monitored by the execution environment for successful termination or for exception failures. The module provides a recovery mechanism to maintain non-stop service execution in the presence of faults.

The decision module consists of a set of detection elements that examine the output (state changes) generated by individual executions, process the results, and reach a decision. The module must be invoked whenever parallel running instances of services reach predefined points in their execution, or when an exception is generated.

In the remainder of this section, we outline the functional requirements for these two modules.

2.1 Execution Module

The execution module designates the mechanism for the application of patches to service instances. Ideally, the patching mechanism should be able to: (*i*) apply patches to running service instances with little or no down-time, (*ii*) allow for the application of arbitrary patches (*i.e.,* not just binary-compatible), (*iii*) have the ability to apply patches even in the absence of source code, (*iv*) provide the ability to insert multiple patches at specific program locations, and (*v*) monitor multiple speculative executions for successful completion or faults.

There are a few instrumentation injection techniques that can be employed for the purposes of our approach.

- Binary translation, where the tool adds instrumentation to a compiled binary, *e.g.,* ATOM [15]
- Runtime instrumentation, where the code is instrumented just prior to execution, *e.g.,* PIN [7]
- Runtime injection. This is a more light-weight approach than runtime instrumentation, where the code is modified at runtime by inserting jumps to helper functions, *e.g.,* dyninst [3]

Runtime instrumentation provides the most flexible platform, at the cost of higher performance overhead. For example PIN [7], a runtime injection tool, uses a dynamic translation to intercept and make changes to runtime code, which typically adds a $2x$ overhead. Dyninst [3] uses a runtime injection approach that adds minimal overhead to the application but requires that patches maintain binary compatibility. The issue of binary compatibility is of minor importance given the fact that the majority of vulnerability patches usually involve minimal changes to the underlying source code. In fact, in order to provide any sort of type-safety on dynamic software updating, one would have to use a language that is dynamic-update aware [16]. The overhead of executing the inserted instrumentation is comparable to that of a function call and primarily depends on how efficiently registers can be saved and restored during execution.

Numerous approaches on supporting dynamic software updating (DSU) have been proposed in the literature [5, 2, 10]. The most flexible, in terms of the kind of updates that are supported, are popcorn [5] and ginseng [10]. Both of these systems use a compiler-based approach to ensure type-safety and data integrity for software updates. In ginseng, source-to-source transformations are used to handle the transformation of data whose type changes as a result of the update and to allow for the dynamic update of infinite loops. Unfortunately, the cost of update flexibility is performance. Ginseng exhibits a performance overhead that is in the order of 30%. Furthermore, since the base application needs to be compiled using the ginseng system prior to software updates deployment, this performance overhead is paid at all times. Ideally, a system should suffer no performance overhead when patches are not being tested.

While the ability to support complex software updates will always require complex DSU systems to ensure update safety, support for simple security patches is less involved. The majority of released security patches are composed of

minimal changes to the logic of the application, often consisting of single-line changes to the source code. This type of update is easily handled using runtime injection, since it typically maintains binary compatibility at the function level.

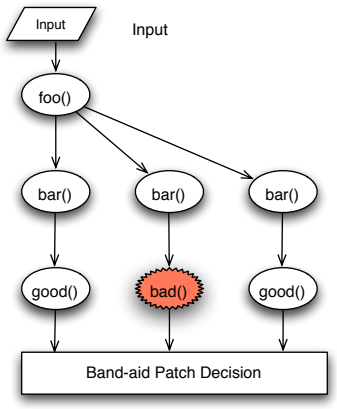

Figure 1. Overview: The service is logically replicated and multiple instances run concurrently in the execution environment. At the end of executing the patches, the system must decide which execution path to use.

2.2 Decision Module

At a high level, the decision module is responsible for detecting variations, and subsequently violations of execution state. Determining the correctness of the resulting execution is based on a combination of policy and heuristics. Working under the assumption that the base implementation is an instance of "correct" execution, we can derive deviations from this model as potentially problematic. The following two example scenarios illustrate the range of policies that can be used by our approach:

1. *Correct base instance*: In this context, the assumption is that the base implementation defines the measure of "correctness" for the system. There are numerous ways to determine correctness, one being empirical observation. For example, telephony switches are periodically retrofitted with vulnerability patches that should not alter their normal operation (*e.g.*, how calls are routed). Patches to such systems must characterize functional deviation as potentially problematic.

2. *Well defined end-state*: In cases where well defined invariants and specifications of the result state are available, execution output can be checked against those constraints. For example, database systems have well-defined data constraints, such as the range of values in fields.

Several mechanisms can be combined as part of our Band-aid Patch architecture. Our goal is to identify differences in state that might occur during different executions and to detect violations to system policy as defined above. The two primary candidates for measuring deviation in execution state are memory traces and I/O transactions.

A number of techniques can be applied to memory views generated by the different executions depending on the characteristics of the application and the granularity of information we are trying to extract. Comparing the full memory layout is the simplest approach, but generates copious amounts of information. Context-aware memory "diffs" can be used to filter out superfluous memory information. Memory management at the library/system-call level [14] can be used to fingerprint each execution. Finally, measuring variation in inter/intra–function call-graphs can provide additional hints on the effects of program semantics [1].

When considering I/O transactions, the basic concept is to fork filesystem views for each execution. At completion time, it is possible compare file system views for inconsistencies, *e.g.,* using mechanisms such as the versioning file system [9]. Discrepancies and inconsistencies can be handled as indications of anomalous execution. Similarly, for network I/O, operations can be filtered and compared with the anticipated behavior, defined by the appropriate policy or by an external anomaly detector.

The decision component can be injected at specific locations in the code, similar to the patch insertion described earlier. Figure 1 illustrates the concept. In the example, the execution module is invoked at function `foo()`, allowing for the simultaneous deployment of two patches. At that time, two more instances of the service are created. Execution continues normally for all instances until we reach (*i*) a predefined point in the execution or (*ii*) an exception is raised.

3 Exploratory Prototype

To determine the feasibility and identify technical hurdles of our proposed approach, we developed a preliminary prototype. For this particular implementation, we chose to use runtime injection for patch insertion. We use dyninst [3] due to its low runtime overhead and its ability to attach and detach from already running processes.

The updates supported by our prototype happen at function-level granularity, and come into effect on the next invocation of the replaced function. The different versions of the patches to be tested are created as a dynamic library

that can be linked to the application at runtime. The function where the Band-aid Patching will be initiated in is instrumented to include calls to the patches to be tested on function entry.

Unfortunately, the obvious choice of "forking" a different process for each execution threat carries the stigma associated with changing process information; to avoid breaking program semantics, special care needs to be placed on programs that rely on process ID information. Using dyninst, we can intercept calls to `getpid` and its derivatives to return the appropriate process ID. However, this solution is no panacea as the process ID might have been communicated to other parts of the application prior to the patch deployment. A possible solution is to add a thin virtualization layer that maps all process IDs to virtual values [11]. An alternative approach would be to execute the different versions of a patch in succession. While this approach is elegant in its simplicity, it means that we can only explore the effects of a patch within the confines of the function, *i.e.,* we would not be able to examine possible side-effects exhibited by a patch farther down in program execution. We also cannot take advantage of hardware facilities, such as multiple processors/cores, that could minimize the overhead of our scheme. For this particular implementation we employ the sequential patch approach using a tool we have previously developed, STEM [13]. The decision component is currently implemented as an application-specific component that can be injected at particular program locations using the mechanism described previously. For our prototype, we use STEM to detect general faults and memory violations. If a patch does not introduce a failure during execution, execution is allowed to continue. If the patch causes a failure, execution continues with the un-patched version of the code.

4 Conclusions

We have proposed a new approach for dynamically testing multiple patches on long-running service instances. To gain some insight into the issues associated with our approach, we have developed a prototype implementation. This proof-of-concept system has addressed and identified a set of practical issues pertaining to this new approach. However, we feel that this set of issues represents only the tip of the iceberg. Further research will need to focus on these pressing issues like gaming attacks, semantic correctness and efficient state comparisons.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the* 12^{th} *ACM conference on Computer and Communications Security (CCS)*, pages 340–353, 2005.

[2] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz. OPUS: Online Patches and Updates for Security. In *Proceedings of the* 14^{th} *USENIX Security Symposium*, August 2005.

[3] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[4] C. Cowan, H. Hinton, C. Pu, and J. Walpole. The Cracker Patch Choice: An Analysis of Post Hoc Security Techniques. In *Proceedings of the National Information Systems Security Conference (NISSC)*, October 2000.

[5] M. W. Hicks, J. T. Moore, and S. Nettles. Dynamic Software Updating. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–23, June 2001.

[6] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting Past and Present Intrusions through Vulnerability-Specific Predicates. In *Proceeding of the ACM Symposium on Operating Systems Principles (SOSP)*, October 2006.

[7] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, pages 190–200, June 2005.

[8] E. Marcus and H. Stern. *Blueprints for high availability: designing resilient distributed systems*. John Wiley & Sons, Inc., 2000.

[9] K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A Versatile and User-Oriented Versioning File System. In *Proceedings of the* 3^{rd} *USENIX Conference on File and Storage Technologies (FAST)*, pages 115–128, March/April 2004.

[10] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for c. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 72–83, 2006.

[11] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: A system for migrating computing environments. In *Proceedings of the* 5^{th} *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 361–376, December 2002.

[12] E. Rescorla. Security holes... Who cares? In *Proceedings of the 12th USENIX Security Conference*, August 2003.

[13] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a Reactive Immune System for Software Services. In *Proceedings of the USENIX Technical Conference*, pages 149–161, April 2005.

[14] A. Somayaji and S. Forrest. Automated response using System-Call delays. In *Proceedings of the* 9^{th} *USENIX Security Symposium*, pages 185–198, August 2000.

[15] A. Srivastava and A. Eustace. Atom: a system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 196–205, June 1994.

[16] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: safe and predictable dynamic software updating. In *Proceedings of the* 32^{nd} *ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pages 183–194, 2005.