Poster: Targeted Therapy for Program Bugs

Qiang Zeng
PhD Candidate
Penn State University
qzeng@cse.psu.edu

Mingyi Zhao
PhD Candidate
Penn State university
muz127@ist.psu.edu

Peng Liu
Professor
Penn State university
pliu@ist.psu.edu

***Abstract*—Program bugs widely exist and render software faulty and vulnerable. Existing systems for surviving software failures and attacks are mostly like chemotherapy, a cancer therapy that causes severe adverse side effects because of imprecise treatments. We propose *Software Targeted Therapy*, a new model for surviving software failures and attacks due to program bugs, that characterizes cancer-cells-like program elements concisely at diagnosis and then applies treatments to them precisely at software execution with minimal overhead. As a case study, we apply Software Targeted Therapy to addressing heap buffer overflows, one of the most dangerous software vulnerabilities, and implement *HeapTherapy*. To our knowledge, HeapTherapy is the first efficient design that prevents memory corruption caused by buffer overflows without disrupting software execution (e.g., bounds checking can only prevent corruption). The slowdown averages 6% on SPEC CPU2006 when HeapTherapy treats up to 10 heap buffer overflow bugs simultaneously.**

I. Introduction

Bill Gates said "Microsoft products are generally bug free," while in practice both their operating systems and service programs contain a large number of bugs. Program bugs are one of the main reasons of software failures, which degrade the reliability and availability of services and lead to potential denial-of-service attacks.

Moreover, with the wide deployment of some defense techniques especially randomizations, such as ASLR and random canaries, the attacker resorts to means of obtaining critical information through brute-force or other more effective attacks through repetitively exploiting specific vulnerabilities. For example, BROP (Blind ROP) [1] introduces a generic stack reading attacking technique, which guesses the value of a single byte of needed information (canaries, saved frame pointers and return addresses) at each buffer overflow attack and detects a hit of the correct value if the victim service does not crash. Such attacking technique is very effective, for it only needs 128 tries on average to leak one byte of information. Thus, a successful return oriented programming attack can be constructed based on the gathered information after crashing the victim service thousands of times.

Therefore, systems that can respond automatically upon an attack or software failure and prevent both denial-of-service attacks and repetitive exploitations of a vulnerability as used in brute-force and BROP attacks are demanded. Many approaches have been proposed to survive software failures and attacks. N-version and N-variant systems leverage software diversity to survive failures [2]. They rely on synchronization between redundant executions, and thus incur high overhead throughout the life cycle of the system. Rx system adjusts the program execution environment when a failure occurs [3]. It lacks precise information guiding such adjustments; instead, it follows some intuition-based rules to try various adjustments until one works. The trial-and-error approach may experience many failures before a success. Some systems generate signatures of user requests that ever caused failures, in order to filter out those abnormal requests at new runs [4], [5]; they have false positives and become less effective when handling requests containing polymorphic malicious code. Some systems disrupt the development of a failure by forcing a function return or returning crafted bytes for overrun read; they share the limitation of semantic deviation [6], [7]. *The common limitation of most prior work is the lack of preciseness.*

We approach from a medical perspective by regarding software as human body, and treat software failures and attacks like treating cancers. Prior work is like chemotherapies for cancer treatments, which act by killing all cells that divide rapidly, one of the main properties of most cancer cells as well some normal cells, and thus cause severe side-effects. Given a bug, prior work also applies enhancements to irrelevant program parts.

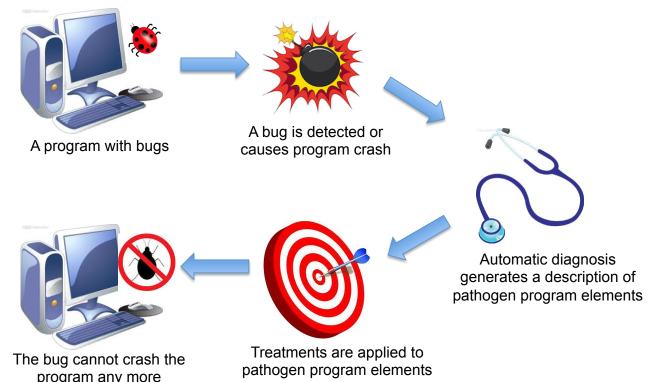

Fig. 1. The life cycle of the detection, diagnosis, and recovery process with Targeted Therapy.

As a contrast to chemotherapy, Targeted Therapy for cancer treatments is a type of more effective medication that blocks the growth of cancer cells precisely. Learning from the spirit of Targeted Therapy, we propose an approach named *Software Targeted Therapy*. As shown in Figure 1, upon the detection of a program bug, the diagnosis characterizes program elements that may be affected by the bug, which we call *pathogen* elements. At the recovered execution those pathogen program elements are identified efficiently based on the diagnosis result, and enhancements are applied to them precisely. The program execution then is immune from the bug.

II. Description Method of Program Elements

The uniqueness of Software Targeted Therapy is that (1) its diagnosis is able to describe the pathogen program elements concisely at diagnosis and (2) its execution can identify pathogens precisely and efficiently.

We present a description method that characterizes concisely and identifies program elements efficiently. It involves two challenges. First, a program element, such as a buffer, a function call, a thread and a variable, contains a variety of information, e.g., the address, the call path and the calling context; however, it is difficult to identify the piece of information that can be reproduced between program executions. For example, the memory address of a variable varies due to ASLR. Second, the program execution should be able to make use of the description result efficiently.

Definition 2.1: A **sketchy control flow path** of a program location is a tuple consisting of the location's calling context and intra-procedural control flow path. In the rest of the paper, we refer to it as a sketchy path unless stated otherwise.

A simple example of intuition is that when debugging a failure, such as a segmentational fault due to a double free bug, instead of attributing the failure to the current function, we routinely first obtain the calling context as a start point. This inspires us to approach a software failure caused by a bug from a control flow perspective. More importantly, calling context information usually can be reproduced, for example, the calling context of `memcpy` that overruns a buffer. Similarly, take the data race bug as an example, the intra-procedural path in the function containing the data access is critical and usually can be reproduced between executions. So we use the sketchy control flow path of a program element consisting of the calling context and the intra-procedural path as the description target. At diagnosis, the sketchy paths of pathogen program elements are identified through logging or replay.

In order to obtain the calling context at execution, stack walking is straightforward but too inefficient for continuous calling context retrieval. Another approach is to build a dynamic calling context tree where each node in the tree represents a unique context. It incurs times of slowdown, though. A few calling context encoding techniques, which represent a calling context using very few, typically one, integers, have been proposed to track calling contexts continuously with very low overhead ($\leq 3\%$) [8], [9]. While the techniques have been used in profiling, test coverage, and anomaly detection, we propose to use the calling context encoding technique in our description method for the purpose of surviving software failures and attacks.

Ball and Larus proposed an efficient algorithm (referred to as the BL algorithm) to encode intra-procedural paths [10]. The algorithm instruments the program by assigning simple arithmetic operations along the edges of the control flow graph. By executing the arithmetic operations progressively with the function execution, each of the control flows obtains a unique integer encoding. The BL algorithm is used in our description method to encode the intra-procedural path.

Each sketchy path of a pathogen program element is encoded into one or very few integers. We perform a two-tier matching of sketchy paths at execution to identify pathogens. At execution, a continual calling context encoding is performed, so that the encoding value of the current calling context is always available. Only when the current calling context matches the calling context of a pathogen sketchy path, do we perform intra-procedural path encoding and compare it against the pathogen's intra-procedural path to finally determine whether current sketchy path is a pathogen one; if so, a pathogen program element is identified and proper treatments are applied. The two-tier encoding improves scalability, because the overhead due to intra-procedural path encoding is localized. Moreover, for some bugs, e.g., the double free bug, the intra-procedural path is irrelevant, and hence only the calling context information needs encoding.

III. Evaluation Results

We have applied Software Targeted Therapy to building HeapTherapy, which hardens software automatically upon detection of heap buffer overflow attack. Unlike bounds checking and many other buffer overflow countermeasures, which usually stop the program execution upon overrun, HeapTherapy allows the program execution to continue without the risk of memory corruption. It incurs an average of 6% slowdown and 8% memory overhead when effectively defending against up to 10 vulnerabilities simultaneously. Software Targeted Therapy can also be potentially applied to addressing many other bugs and vulnerabilities, such as stack buffer overflows, dangling pointers, and data races.

References

[1] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh, "Hacking blind," in *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[2] 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 2006*, pp. 105–120.

[3] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: Treating bugs as allergies—a safe method to survive software failures," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, 2005, pp. 235–248.

[4] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt, "Automatic diagnosis and response to memory corruption vulnerabilities," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005, pp. 223–234.

[5] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Towards automatic generation of vulnerability-based signatures," in *IEEE Symposium on Security and Privacy*, 2006.

[6] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis, "Building a reactive immune system for software services," in *Proceedings of the general track, 2005 USENIX annual technical conference: April 10-15, 2005, Anaheim, CA, USA*. USENIX, 2005, pp. 149–161.

[7] M. C. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, "Enhancing server availability and security through failure-oblivious computing." in *OSDI*, vol. 4, 2004, pp. 21–21.

[8] M. D. Bond and K. S. McKinley, "Probabilistic calling context," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, 2007, pp. 97–112.

[9] Q. Zeng, J. Rhee, H. Zhang, N. Arora, G. Jiang, and P. Liu, "DeltaPath: Precise and Scalable Calling Context Encoding," in *Symposium on Code Generation and Optimization*, 2014.

[10] T. Ball and J. R. Larus, "Efficient path profiling," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, 1996, pp. 46–57.