Using Diversity to Harden Multithreaded Programs Against Exploitation

David M. Tagatac and Salvatore J. Stolfo
Columbia University
New York, NY, USA
{dtagatac,sal}@cs.columbia.edu

Michalis Polychronakis
Stony Brook University
Stony Brook, NY, USA
mikepo@cs.stonybrook.edu

***Abstract*—Multithreaded programming is here to stay, and concurrency bugs are the focus of a growing number of cyberattacks. While most defensive efforts against such attacks seek to identify bugs during debugging, an alternative method seeks to make exploitation harder without the need to first identify the bugs—or even the fact that there are any. Time randomization introduces more diversity among instances of the same software. In much the same way that ASLR-induced diversity in memory locations thwarts attacks crafted for specific addresses, time randomization-induced diversity in thread timing aims to thwart concurrency attacks crafted for specific vulnerability windows.**

We study three implementations of time randomization, all using the injection of NOPs to alter program timing. Their application to two real-world concurrency bugs results in a marked increase in the cost to exploit those bugs. After demonstrating the effectiveness of the method, especially when NOPs are injected before library function calls following synchronization points, methods for improving the efficiency of this defense against concurrency attacks in future research are proposed.

***Index Terms*—time randomization, concurrency bugs, software diversity.**

I. Introduction

With the pervasiveness of multicore architectures, multithreading is an important—and often necessary—tool when programming for performance. However, programming with multiple threads is generally more difficult than programming for serial execution. Each thread has the potential to contain any bug of a serial program, and on top of that, the uncertain interleaving of concurrent threads has the potential for concurrency bugs (e.g., data races).

Partial taxonomies of concurrency bugs have been constructed [1], [2], and it has been demonstrated that attacks on buggy multithreaded programs are a real concern [3]. Much of the effort in combating this threat has gone into tools and systems which detect data races in order to aid debugging [4], [5], [6], [7], [8]. An alternative approach is to guide multithreaded programs into memoized synchronization schedules [9]. This approach does not dwell on race detection, but rather on removing the nondeterminism from the portions of multithreaded programs where races are most likely. However, schedule memoization in its most automated form is still susceptible to attack whenever the attacker can trigger a different schedule by changing the input.

We address the threat of concurrency attacks from yet another angle: automated software diversity. Software homogeneity is dangerous in that it provides economies of scale to attackers [10]. The relatively high cost of constructing an exploit for one bug is amortized by the opportunity to reuse that same exploit on every identical instance of the software. As a result, automated software diversity is an active area of research for software in general [11]. Concurrency vulnerabilities subject a homogenous world of software to the same dangers for the same reasons.

Focusing on the applicability of diversity-based defenses to concurrency attacks, what we call "time randomization" is the randomization of synchronization schedules and thread interleavings, as well as the relative timing between and among threads. The majority of diversification strategies systematized by Larsen et al. [11], like ASLR and instruction reordering, focus on mitigating attacks which depend on the predictability of absolute and/or relative addresses in memory (e.g., code reuse attacks or stack smashing attacks). Time randomization instead focuses on mitigating attacks which depend on the predictability of relative thread timing. This approach to defenses against concurrency attacks has the advantage that it does not require knowledge of the bug. In other words, time randomization can be applied independent of bug detection and fixing. This highlights another point—that time randomization does not exclude other defenses against concurrency attacks; they can be applied together in concert.

This work makes the following contributions:

- **Time randomization**, a novel approach to concurrency attack defense involving automated software diversity. This work analyzes three implementations which can be considered automated software diversity transformations via functionally-invariant code insertion. All three transformations involve the insertion of NOPs into multithreaded programs, one of which is a mostly unfocused randomization, and two of which specifically target synchronization mechanisms as a heuristic for time randomization. (§ II)
- **Experimental evaluation** of time randomization on two real-world bugs, with encouraging results. In the first bug, the most effective transformation resulted in exploit success rates of less than 0.5% for all experiments beyond a certain threshold of NOPs inserted. In the second bug, the most effective transformation resulted in nearly 85% of exploit attempts taking significantly longer than the

baseline exploit time (measured without time randomization). (§ III and § IV)
- **A proposal for future work** to build on these results, including system-level time randomization for achieving the same benefits with more tolerable overhead. (§ V)

II. Time Randomization

A. Threat Model

The type of attack that time randomization seeks to thwart has the following properties:

- The attack targets a concurrency bug.
- The attacker can gain knowledge about the relative timing between two or more threads relevant to the bug.
- The attacker leverages this knowledge to craft the attack.

Note that there are many ways to gain knowledge about relative thread timing, including results from fuzzing experiments. In the case that the fuzzing is done on a binary, it may not even be obvious how many threads are involved. Nevertheless, positive fuzzing results can be considered knowledge about relative thread timing which can then be used to craft an attack.

As an example, consider the following scenario: A remote attacker is communicating with some server software that has a concurrency bug. They have devised a way to exploit the bug, which includes a method for inducing a buggy thread interleaving. For concreteness, let's assume that the bug is exposed when a save operation is attempted by one thread during the critical section of another save operation in another thread (e.g., after some checking has been done, but before the results have been used, sometimes referred to as a time of check to time of use attack). If the server immediately spawns threads to execute save operations in response to client requests, the attacker's work consists of identifying the proper delay between two save threads such that the critical sections intersect. The critical sections in this context are sometimes referred to as a vulnerability window [3].

If the server software is available to the attacker, they can simply study it on a similar system (which they control) to determine the appropriate delay required to expose the bug. Armed with this knowledge, they stand a good chance of exploiting the bug on the target system by sending requests with the same delay.

B. Implementation

Time randomization, the randomization of synchronization schedules and thread interleavings, as well as the relative timing between and among threads, aims to make this type of attack much harder. By making the relative timing between threads less predictable, the cost of carrying out the attack above is dramatically increased.

Of the various software diversity transformations which have been studied [11], the ones which are most applicable to time randomization are those which

- add delays to a thread or threads, and those which
- apply reordering to affect thread interleaving.

The first category is almost always possible, as adding delays rarely breaks programs. However, an obvious drawback to this approach is loss of performance, and this drawback must be weighed against the benefits of the transformation. In the second case, the "something" reordered can be at any granularity (instruction level, function level, program level, etc.) so long as thread interleaving is affected. These types of transformations are less likely to have performance costs, but they are more likely to break the correctness of programs. This work studies three transformations in the first category.

The three transformations for introducing automated diversity explored in the context of concurrency attacks in this work consist of interposing external library calls [12] made by the software to be protected each with random numbers of inline assembly NOP instructions. This was accomplished in two steps: first by specifying the library functions to be interposed; and second by interposing those functions with NOP loops.

1) Choosing Which Library Functions to Interpose: The way that the library functions are chosen for interposition distinguishes among the three transformations studied. The first transformation chooses all external library function calls indiscriminately, while the other two use synchronization mechanisms to indicate locations where delays may have the greatest effect on relative thread timing. In all transformations, `ltrace` [13] is run on the unmodified program to be protected while it is being subjected to an exploit attempt. Using the log from that ltrace run,

T1 the first transformation chooses every external library function call for interposition,

T2 the second transformation chooses every external library function call immediately preceding a synchronization mechanism, and

T3 the third transformation chooses every external library function call immediately following a synchronization mechanism.

(Hereafter, these transformations are referred to as **T1**, **T2**, and **T3**.) Functions were only excluded in the rare cases when they take a variable argument list, or interposing the function interferes with the interposition method itself (e.g., `sigsetjmp`). Both cases are unsuitable for the interposition method used in this work. As all of the experiments were conducted on Linux, "synchronization mechanisms" were generally identified as `pthread` function calls, except for the case of Libvirt which redefines its own synchronization mechanisms.

2) NOP Loop Injection: Once the functions were chosen for interposition, NOP loops were interposed as described in the tutorial cited above [12]. First a max delay was specified (per interposition). Then, for each of the external library functions chosen for interposition, a random integer between zero and the max delay was selected. Finally, that number was used as the loop limit on a loop of a single assembly NOP interposed before that external library function. The interpositions of the NOP loops were compiled into a dynamic library file, and that file was specified to be preloaded with the LD_PRELOAD environment variable.

More specifically, a C file was generated (including any necessary headers) which redefined each of the library functions

chosen for interposition as described in § II-B1. Each of these redefinitions consisted of exactly the following pieces:

1) declaration of a function pointer of the same type as the function being redefined (to be used to call the original library function),
2) a `for` loop with a loop limit chosen as described above, and a body consisting of the single statement `asm("nop;");`,
3) assignment of the original library function (via `dlsym`) to the function pointer declared in 1., and
4) a call to the original library function, returning the return value.

This file was then compiled as a dynamic library to be preloaded at runtime before each experiment.

III. Experimental Design

To test both the efficacy and the performance of time randomization, we applied time randomization to two real concurrency bugs.

A. Libsafe CVE-2005-1125

Libsafe [14] is a library which protects processes against the exploitation of buffer overflow vulnerabilities in process stacks, as well as format string vulnerabilities. It does this by intercepting all calls to C standard library functions known to be vulnerable, and then running "safe" versions of those functions which issue warnings about exploit attempts before exiting without allowing the exploits to succeed.

A static global variable 'dying' is used in Libsafe to indicate when an unsafe action has already been detected, and Libsafe is issuing warnings and exiting. In this case, Libsafe stops checking for new exploit attempts. However, this variable is not protected by any synchronization mechanisms, so in the case of a multithreaded program, the following sequence of events is possible:

1) Thread A attempts an exploit.
2) The exploit is caught by Libsafe, the 'dying' flag is set, and Libsafe begins the warning and exit procedure.
3) Thread B is scheduled, and attempts another exploit before Libsafe has exited thread A.
4) Because the 'dying' flag is set, thread B's exploit is not caught by Libsafe, and it succeeds.

The proof-of-concept exploit [15] often realizes the above sequence of events, effectively bypassing Libsafe.

Time randomization was applied to Libsafe as described in § II-B. The time required to make a (legal) `strcpy` function call (using Libsafe and averaged over 10,000,000 `strcpy` calls), and the proof of concept exploit success rate were measured for several randomizations for each max delay, where max delay was varied from 0 to 50,000. Recall that the max delay is used as the upper bound on the range from which the NOP loop length for each function interposition is randomly chosen, as described in § II-B. **T2** and **T3** were special cases for this bug in that only one external library function was identified as immediately preceding a synchronization mechanism and only one external library function was identified as immediately following a synchronization mechanism. In these cases, we were able to exactly characterize the effects of NOP injection as a function of the number of NOPs injected into that function.

B. Libvirt CVE-2014-1447

Libvirt [16] is a toolkit for interacting with virtual machines and hypervisors. This toolkit consists, in part, of a daemon 'libvirtd' which listens for and responds to virtual machine management requests (e.g., to connect to a remote host, or shutdown a VM).

Prior to version 1.2.1, the Libvirt daemon suffered from concurrency bug CVE-2014-1447 which allows an unauthenticated client to crash the daemon with seemingly innocuous requests to connect to a host. After receiving a request from a client to open a connection, the daemon spawns a thread to open and maintain that connection. The spawned thread will eventually dereference a pointer called `client->keepalive` as part of opening the connection. However, during this vulnerability window, the client may close the connection, which the daemon will handle in the main thread by (among other things) setting `client->keepalive` to `NULL`. Then when the spawned thread dereferences this pointer, the daemon crashes. [17]

The Libvirt developers' proof of concept makes two changes to Libvirt in order to reliably reproduce this bug. First, the client is modified to exit as soon as it has requested a connection. Second, the daemon is modified to sleep in threads spawned to open new connections, before dereferencing the `client->keepalive` pointer. The first modification is within an attacker's ability, in our threat model. However, the second modification requires control of the daemon, which we assume the attacker does not yet have. An alternate exploit requiring only the first modification was applied to Libvirt 0.9.8. In this alternate exploit, the client opens (and closes) as many connections as it can until the daemon crashes. Without modification to the daemon, this alternate exploit reliably reproduced the bug and crashed the daemon.

Time randomization was applied to the Libvirt daemon as described in § II-B. The time required to open and close a connection properly (with the unmodified client averaged over 10,000 sequential connections) was measured, and the time required for the alternate exploit described above to crash the daemon was measured five times for a single randomization for each max delay. For **T1**, the max delay was varied from 0 to 1,000. For **T2** and **T3**, the max delay was varied from 0 to 1,000,000. All exploit attempts were interrupted and considered failed attempts after ten minutes if they did not succeed before then.

IV. Results

We evaluated **T1**-**T3** on Libsafe CVE-2005-1125 and on Libvirt CVE-2014-1447. For each evaluation, exploit cost or exploit success rate was evaluated for each of many applications of the transformation under study. The exploit cost or success rate was then compared to either the microbenchmark

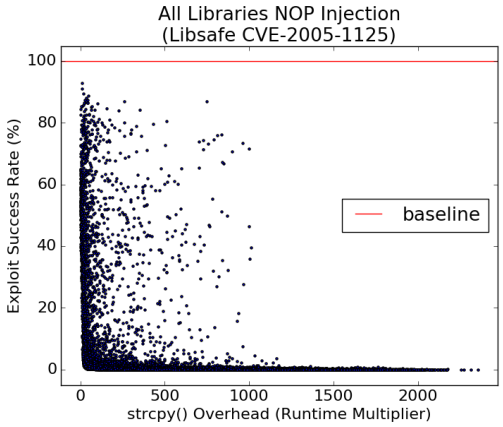

Fig. 1: Exploit success rate as a function of the microbenchmark after applying **T1** to Libsafe with concurrency bug CVE-2005-1125. There is a clear inverse correlation between microbenchmark overhead and exploit success rate.

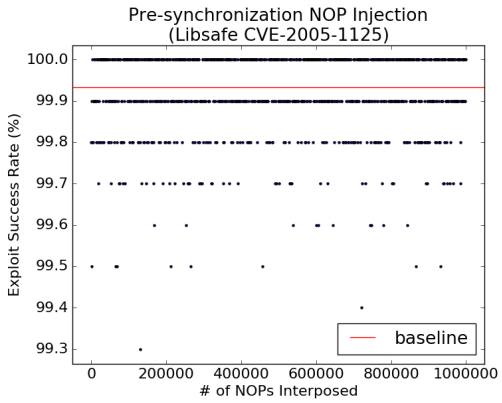

Fig. 2: Exploit success rate as a function of the microbenchmark after applying diversity **T2** to Libsafe with concurrency bug CVE-2005-1125. This transformation does not appear to have an effect on exploit success rate for this bug.

overhead, or the number of NOPs injected, as appropriate for each experiment. Libsafe experiments were run on a dedicated dual core 3.4GHz Pentium D running Debian. Libvirt experiments testing **T1** were run on a 2.7GHz dual-socket quad-core Intel Xeon with 16 hyper-threading cores running Debian. Libvirt experiments testing **T2** and **T3** were run on a 2.8GHz dual-socket hex-core Intel Xeon with 24 hyper-threading cores running Ubuntu. In all figures, the horizontal red line indicates the average measurement for the unmodified bug (without time randomization). The results vary somewhat widely with the bug analyzed and the automated diversity transformation used.

Applying transformation 1 to Libsafe resulted in a dramatic decrease in exploit success rate with increasing microbenchmark overhead (Fig. 1). Between 1x and 12x overhead, no randomizations resulted in less than a 5% success rate. Between 20x and 25x overhead, 27.5% of randomizations resulted in less than a 5% exploit success rate. Between 40x and 45x overhead, 61.5% of randomizations resulted in less than a 5% exploit success rate. Beyond 1,040x overhead, 100% of randomizations resulted in less than a 5% exploit success rate.

Applying **T2** to Libsafe had no noticeable effect on exploit success rate (Fig. 2) or microbenchmark overhead.

Applying **T3** to Libsafe had a very marked effect on exploit success rate (Fig. 3). In particular, the rate remains high for low numbers of NOPs. Then, for inserted NOP loops of lengths greater than 200,000, the rate begins to drop. For NOP loops of lengths between 250,000 and 350,000, exploit success rate fluctuates between 35% and 65%. Finally, for NOP loops of lengths greater than 350,000, exploit rates drop even further, and beyond loop-lengths of 390,000 NOPs, all interpositions resulted in exploit rates less than 0.5%.

T1 affected the exploit cost for the Libvirt bug. However, the increase in time required for exploitation is approximately proportional to the increase in microbenchmark measurements. The exploit cost increase may therefore be attributed to an overall slowing down of the Libvirt daemon (Fig. 4).

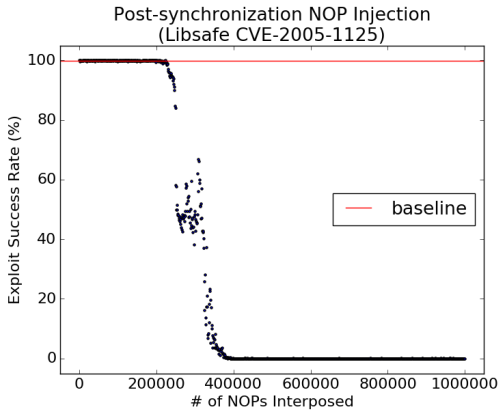

Fig. 3: Exploit success rate as a function of the number of NOPs injected after applying **T3** to Libsafe with concurrency bug CVE-2005-1125. A very clear inverse correlation between NOP loop length and exploit success rate is shown.

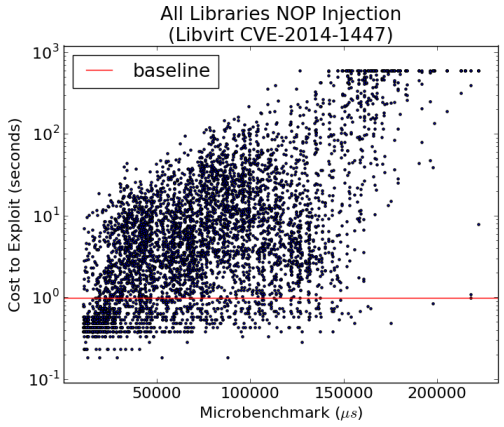

Fig. 4: Exploit cost (in time) as a function of the microbenchmark after applying **T1** to Libvirt with concurrency bug CVE-2014-1447. The exploit cost increases as a result of applying this transformation, but the increase is approximately proportional to the overall performance loss (x-axis).

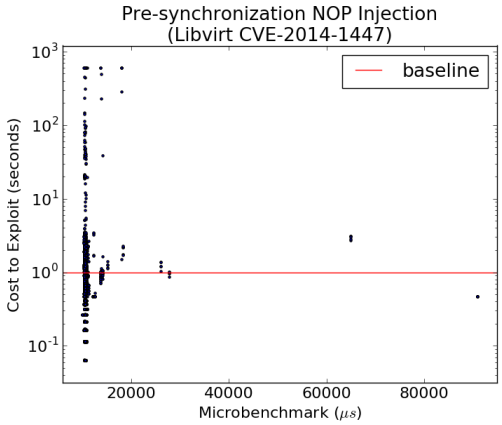

Fig. 5: Exploit cost (in time) as a function of the microbenchmark after applying **T2** to Libvirt with concurrency bug CVE-2014-1447. This transformation does not appear to have an effect on exploit success rate for this bug.

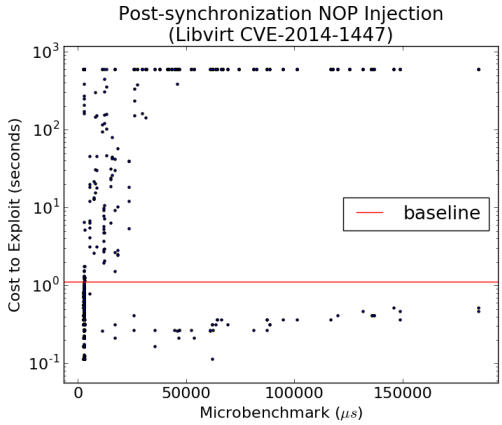

Fig. 6: Exploit cost (in time) as a function of the microbenchmark after applying diversity **T3** to Libvirt with concurrency bug CVE-2014-1447. Beyond microbenchmark measurements of 10 milliseconds, most of exploit costs observed were much greater than the baseline; often the exploit attempts were stopped artificially early and labeled "failed exploits." This transformation thus increases the average exploit cost with increasing microbenchmark overhead.

Applying **T2** to Libvirt did not seem to have much of an effect on exploit cost. The characteristic of most exploit attempts succeeding quickly, and a few taking much longer appears to be consistent throughout the range of microbenchmark values recorded (Fig. 5).

T3 applied to Libvirt, as in the case of Libsafe, had a significant effect on exploit cost. For microbenchmark connection times of less than 10 milliseconds, less than 4% of exploit attempts took longer than the baseline exploit time of 1.11 seconds, where the baseline exploit time was obtained for a Libvirt daemon without any time randomization. By contrast, for microbenchmark times of greater than 10 milliseconds, nearly 85% of exploit attempts took longer than the baseline exploit time. Looking at the graph (Fig. 6), we can see a similar shape to the one obtained when applying **T2** to Libvirt - the bifurcation between exploit attempts that succeed quickly within seconds, and those which take hundreds of seconds to succeed. However, after applying automated **T3**, a significantly greater proportion of exploit attempts fell into the second branch of that bifurcation, taking hundreds of seconds to succeed, or failing to succeed within ten minutes (and being categorized as failed attempts).

V. Discussion & Future Directions

It is clear from these results that, at the very least, time randomization increases the exploitation cost for some concurrency bugs. **T3** appears to be more effective than the other two transformations. One possible explanation for this is that **T3** adds delays shortly after synchronization mechanisms. If these delays fall in a critical section, or before one, the chances of scheduler preemption affecting the thread interleaving about the critical section increase. This is not to say that the other transformations cannot also have this effect, but in the case of **T2**, the proximity to a critical section may not be as high, and in the case of **T1**, the delays are less concentrated about the synchronization mechanisms.

Given the results presented here, the major argument against employing time randomization universally becomes performance overhead. While overheads on effective randomizations were unacceptably high using the NOP injection implementations tested here (at least 12x for **T1** and at least 5x for **T3**), it is not clear that significant overhead is necessary for time randomization in general. Part of the effect of time randomization via NOP injection is to modify thread interleavings by signaling with NOPs to the scheduler that another thread can be scheduled. The randomization between and among threads achieved here with NOP injection could conceivably be achieved directly in the scheduler via random synchronization schedules. By modifying the scheduler directly, it may be possible to obtain the same benefits shown here with minimal overhead. In this vein, it is also worth evaluating established and novel scheduling algorithms which include elements of randomness (e.g. genetic scheduling algorithms [18]).

Alternatively, a more complicated heuristic could be applied to the NOP insertion implementations investigated here. More specifically, for each randomization, measurements could be taken of the extent to which relative thread timing has been randomized, and of some performance metric. These measurements could be used to decide whether to keep a specific randomization or generate a new one, as well as to calibrate randomization parameters like max delay in this work.

VI. Related Work

Automated software diversity as a defense strategy has been systematized by Larsen et al. [11]. That work proposes a taxonomy of cyberattacks and a taxonomy of defenses against those classes of attacks, and surveys many of the studied and unstudied diversification techniques. To our knowledge, ours is the first work to investigate automated software diversity specifically as it pertains to concurrency attacks.

NOP injection, also known as "garbage code insertion," has been investigated as a defense strategy many times. Cohen [19] and Forrest et al. [20] proposed garbage code insertion as a method for diversifying operating system source code, aiming to drive up the required complexity of attacks. Onarlioglu et al. investigated NOP injection as a means to create gadget-less binaries [21]. Their approach is deterministic and a distinct defense from automated diversity. Jackson et al. used GCC or LLVM to insert NOPs into register transfer language or machine code, respectively, at compile time [22]. This diversifying transformation inserts a NOP—with some finite probability–before each instruction, in order to break return-oriented programming and code reuse attacks. Homescu et al. went on to measure the performance and efficacy of traditional [23] and JIT [24] compiler-based automated diversity. While the literature for NOP injection as a defense is voluminous, we again believe our work to be the first to investigate it with respect to concurrency attacks. Moreover, where compiler-based NOP insertion requires recompiling entire programs, our approach is more lightweight and amenable to more frequent randomization (e.g., at every reboot).

Occasionally, randomized scheduling has been used to *expose* concurrency bugs. CalFuzzer [25] is a testing framework that allows for pausing of Java programs at critical "breakpoints," and then testing the various possible transitions from those breakpoints for concurrency bugs. Burckhardt et al. showed that random insertion of priority change points in a priority-based scheduler has a probability of exposing concurrency bugs that is reasonably high when it can be assumed that the number of scheduling constraints required to reproduce the bug is low [26]. CTrigger [27] inserts artificial synchronizations at critical execution points in an attempt to trigger low-probability thread interleavings. While these works appear at first glance to contradict time randomization as a defense, this is not the case. Time randomization does not propose to fix existing concurrency bugs, but rather to raise the cost to an attacker attempting to trigger, and then exploit a bug using timing information that they have gained about the bug. Randomization may at times uncover concurrency bugs that don't usually manifest under normal execution, while also hardening the program against targeted concurrency attacks.

VII. Conclusion

We have presented time randomization, a subset of automated software diversity which randomizes the timing between and among threads, with the goal of thwarting concurrency attacks which benefit from predictability in that timing. Our experiments show that at least one of the time randomization transformations tested have a significant effect on the cost to exploit two real-world concurrency bugs. We believe that transformations with lesser performance costs are worth studying as a defense against concurrency attacks.

VIII. Acknowledgment

This work was supported in part by IARPA STONESOUP contract FA8650-10-C-7024.

References

[1] E. Farchi, Y. Nir, and S. Ur, "Concurrent bug patterns and how to test them," *Proceedings International Parallel and Distributed Processing Symposium*, vol. 00, no. C, 2003.

[2] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes - A Comprehensive Study on Real World Concurrency Bug Characteristics," in *ASPLOS XIII*. New York, New York, USA: ACM Press, 2008, p. 329.

[3] J. Yang, A. Cui, S. J. Stolfo, and S. Sethumadhavan, "Concurrency Attacks," in *HotPar'12*. USENIX Association, Jun. 2012, p. 15.

[4] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems*, vol. 15, no. 4, pp. 391–411, Nov. 1997.

[5] C. Flanagan and S. Freund, "Atomizer: a dynamic atomicity checker for multithreaded programs (summary)," *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pp. 269–271, 2004.

[6] O. Laadan, C.-C. Tsai, N. Viennot, C. Blinn, P. S. Du, J. Yang, and J. Nieh, "Finding Concurrency Errors in Sequential CodeOS-level, In-vivo Model Checking of Process Races," in *HotOS XIII*. USENIX Association, 2011, pp. 1–5.

[7] P. Pratikakis, J. S. Foster, and M. Hicks, "LOCKSMITH: Practical Static Race Detection for C," *ACM Transactions on Programming Languages and Systems*, vol. 33, no. 1, pp. 1–55, Jan. 2011.

[8] B. Kasikci, C. Zamfir, and G. Candea, "RaceMob: Crowdsourced Data Race Detection," in *SOSP '13*. New York, New York, USA: ACM Press, 2013, pp. 406–422.

[9] H. Cui, J. Wu, C.-C. Tsai, and J. Yang, "Stable Deterministic Multithreading through Schedule Memoization," in *OSDI'10*. USENIX Association, Oct. 2010, pp. 1–13.

[10] D. Geer, R. Bace, P. Gutmann, P. Metzger, C. Pfleeger, J. Quarterman, and B. Schneier, "Cyber insecurity: The cost of monopoly," *CCIA*, vol. 24, 2003.

[11] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK : Automated Software Diversity," *IEEE S&P*, 2014.

[12] J. Conrod, "Tutorial: Function Interposition in Linux," http://jayconrod.com/posts/23/tutorial-function-interposition-in-linux, Jun. 2009, accessed: 2014-10-04.

[13] J. Cespedes, "Ltrace home page," http://www.ltrace.org/.

[14] T. Tsai, T. Tsai, N. Singh, N. Singh, A. Labs, A. Labs, M. Hill, and M. Hill, "Libsafe 2.0: Detection of Format String Vulnerability Exploits," pp. 1–5, 2001.

[15] "Overflow.pl: Libsafe - Safety Check Bypass Vulnerability," http://www.securityfocus.com/archive/1/395999, Apr. 2005, accessed 2014-10-22.

[16] Red Hat, "libvirt: The Virtualization API," http://libvirt.org, accessed: 2015-01-24.

[17] "Red Hat Bugzilla — Bug 1047577," https://bugzilla.redhat.com/show_bug.cgi?id=1047577, accessed: 2015-01-24.

[18] M. Qiu, M. Zhong, J. Li, K. Gai, and Z. Zong, "Phase-Change Memory Optimization for Green Cloud with Genetic Algorithm," *IEEE Transactions on Computers*, vol. 64, no. 12, pp. 1–1, 2015.

[19] F. B. Cohen, "Operating system protection through program evolution," *Computers & Security*, vol. 12, pp. 565–584, 1993.

[20] S. Forrest, a. Somayaji, and D. Ackley, "Building diverse computer systems," *HotOS VI*, 1997.

[21] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-Free," *ACSAC '10*, p. 49, 2010.

[22] T. Jackson, A. Homescu, S. Crane, P. Larsen, S. Brunthaler, and M. Franz, "Diversifying the Software Stack Using Randomized NOP Insertion," in *Moving Target Defense II: Application of Game Theory and Adversarial Modeling*, 2013, pp. 151–173.

[23] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," *CGO '13*, 2013.

[24] A. Homescu, S. Brunthaler, and M. Franz, "librando : Transparent Code Randomization for Just-in-Time Compilers Categories and Subject Descriptors," *CCS'13*, pp. 993–1003, 2013.

[25] P. Joshi, M. Naik, C. S. Park, and K. Sen, "CalFuzzer: An extensible active testing framework for concurrent programs," in *Lecture Notes in Computer Science*, vol. 5643 LNCS, 2009, pp. 675–681.

[26] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," *ACM SIGPLAN Notices*, vol. 45, no. 3, p. 167, 2010.

[27] S. Park, S. Lu, and Y. Zhou, "CTrigger," p. 25, 2009.