
# Bunshin: Compositing Security Mechanisms through Diversification

**Meng Xu, Kangjie Lu, Taesoo Kim, and Wenke Lee,** ***Georgia Institute of Technology***

https://www.usenix.org/conference/atc17/technical-sessions/presentation/xu-meng

**This paper is included in the Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17).**

**July 12–14, 2017 • Santa Clara, CA, USA**

ISBN 978-1-931971-38-6

**Open access to the Proceedings of the 2017 USENIX Annual Technical Conference is sponsored by USENIX.**

# BUNSHIN: Compositing Security Mechanisms through Diversification

Meng Xu, Kangjie Lu, Taesoo Kim, Wenke Lee
*Georgia Institute of Technology*

## Abstract

A number of security mechanisms have been proposed to harden programs written in unsafe languages, each of which mitigates a specific type of memory error. Intuitively, enforcing multiple security mechanisms on a target program will improve its overall security. However, this is not yet a viable approach in practice because the execution slowdown caused by various security mechanisms is often non-linearly accumulated, making the combined protection prohibitively expensive; further, most security mechanisms are designed for independent or isolated uses and thus are often in conflict with each other, making it impossible to fuse them in a straightforward way.

In this paper, we present BUNSHIN, an N-version-based system that enables different and even conflicting security mechanisms to be combined to secure a program while at the same time reducing the execution slowdown. In particular, we propose an automated mechanism to distribute runtime security checks in multiple program variants in such a way that conflicts between security checks are inherently eliminated and execution slowdown is minimized with parallel execution. We also present an N-version execution engine to seamlessly synchronize these variants so that all distributed security checks work together to guarantee the security of a target program.

## 1 Introduction

Memory errors in programs written in unsafe languages (e.g., C/C++) have been continuously exploited by attackers [14]. To defeat such attacks, the security community has deployed many security mechanisms such as widely deployed W⊕X, which prevents code injection attacks by making Writable memory not eXecutable, and ASLR, which prevents attacks (e.g., code reuse) by making the address of target code/data unpredictable. However, recent attacks [34, 35] have shown that these mechanisms are not difficult to bypass. As such, more advanced techniques have been proposed. For example, SoftBound [28], CETS [29], and AddressSanitizer [33] (ASan) provide a high memory safety guarantee, CFI [1] and CPI [25] effectively mitigate control flow hijacking attacks, MemorySanitizer [36] (MSan) can mitigate information leaks caused by uninitialized read, and UndefinedBehaviorSanitizer [27] (UBSan) can detect the causes of undefined behaviors (e.g., null pointer dereference).

However, despite the large number of software hardening techniques proposed, few of them actually get adopted in practice. One reason is that the slowdown imposed by these mechanisms erases the performance gains that come from low-level languages. Another reason is that each proposed technique tends to fix only specific issues while leaving the program vulnerable to other attacks. Comprehensive security protection is often demanded by mission-critical services such as web servers or cyber-physical systems in which a single unblocked attack could lead to disastrous consequences (e.g., heartbleed [16]).

In order to achieve comprehensive program protection, an intuitive method is to combine several techniques and enforce them together in a target program. Unfortunately, this is often not viable in practice for two reasons: 1) Runtime slowdown increases unpredictably after fusing different techniques. For instance, in an already highly optimized build [29], combining Softbound and CETS yields a 110% slowdown–almost the sum of each technique individually; 2) Implementation conflicts prevent direct combination because most techniques are not designed with compatibility in mind. For instance, MSan makes the lower protected area inaccessible, while ASan reserves the lower memory as shadow memory. Re-implementing these techniques for better compatibility requires significant engineering effort if it is even possible.

In the meantime, hardware is becoming cheaper and more powerful. The increasing number of CPU cores combined with larger cache and memory size keeps boosting the level of parallelism, making it practical to improve software security through a technique known as N-version programming [5, 9, 20, 44]. As part of this trend, the N-version scheme is particularly suitable to multi-core architectures because replicas can run on cores in parallel.

An N-version system typically requires careful construction of N variants that are both functionally similar in normal situations and behaviorally different when under attacks. Hence, although each program version may be vulnerable to certain types of attacks, the security of the whole system relies on the notion that an attacker has to simultaneously succeed in attacking all variants in order to compromise the whole system. This property of the N-version system gives us insight on how to provide strong security to a program and yet not significantly degrade its end-to-end performance. That is, by distributing the intended security to N program variants and synchronizing their execution in parallel, we can achieve the same level of security with only a portion of its running time plus an overhead for synchronization. Hence, the challenges lie in how to produce the program variants in a principled way and how to synchronize and monitor their executions efficiently and correctly.

In this paper, we introduce BUNSHIN, an N-version-based approach to both minimize the slowdown caused by security mechanisms and seamlessly unify multiple security mechanisms without re-engineering efforts to any individual mechanism. In short, BUNSHIN splits the checks required by security mechanisms and distributes them to multiple variants of a program in automated and principled ways to minimize execution slowdown. By synchronizing the execution of these variants, BUNSHIN guarantees comprehensive security for the target program.

While the N-version mechanism has been well studied primarily for fault-tolerance [5, 9, 21], BUNSHIN aims at improving a program's security and enabling the composition of multiple security mechanisms with automated protection distribution mechanisms. In addition, BUNSHIN is a practical system as it does not require extra modification to the system or compilation toolchain. BUNSHIN supports state-of-the-art mechanisms like ASan, MSan, UBSan, Softbound, and CETS. We have tested it on a number of C/C++ programs, including SPEC2006, SPLASH-2x, PARSEC benchmarks, Nginx, and Lighttpd web servers. Through three case studies, we show that 1) the slowdown for ASan can be reduced from 107% to 65.6% and 47.1% by distributing the sanity checks to two and three variants, respectively; 2) the slowdown for UBSan can be reduced from 228% to 129.5% and 94.5% by distributing the sub-sanitizers to two and three variants, respectively; and 3) the time overhead for unifying ASan, MSan, and UBSan with BUNSHIN is only 4.99% more than the highest overhead of enforcing any of the three sanitizers alone. In summary, our work makes the following contributions:

- We propose an N-version approach to enable different or even conflicting protection techniques to be fused for comprehensive security with efficiency.
- We present an improved NXE design in terms of syscall hooking, multithreading support, and execution optimization.
- We have implemented BUNSHIN and validated the effectiveness of BUNSHIN's NXE and the protection distribution mechanisms in amortizing the slowdown caused by state-of-the-art security mechanisms.

The rest of the paper provides background information and compares BUNSHIN with related works (§2), describes the design and implementation of BUNSHIN (§3, §4), presents evaluation results (§5), discusses its limitations and improvements (§6), and concludes (§7).

# 2 Background & Related Work

## 2.1 Memory Errors vs. Sanity Checks

Memory errors occur when the memory is accessed in an unintended manner [37]. The number of reported memory errors is still increasing [14] and severe attacks (e.g., heartbleed [16]) exploiting memory errors emerge from time to time. We provide a taxonomy in Table 1 to summarize the errors. In particular, any vulnerability that may change a pointer unintentionally can be used to achieve out-of-bound reads/writes. Use-after-free and uninitialized read are usually caused by logic bugs (e.g., double-free and use-before-initialization) or compiler features (e.g., padding for alignment). Undefined behaviors can be triggered by various software bugs, such as divide-by-zero and null-pointer dereferences.

| Memory Error | Main Causes | Defenses |
|---|---|---|
| Out-of-bound r/w | lack of length check<br>format string bug<br>integer overflow<br>bad type casting | SoftBound [28]<br>ASan [33] |
| Use-after-free | dangling pointer<br>double free | CETS [29]<br>ASan [33] |
| Uninitialized read | lack of initialization<br>data structure alignment<br>subword copying | MSan [36] |
| Undef behavior | pointer misalignment<br>divide-by-zero<br>null pointer dereference | UBSan [27] |

**Table 1:** A taxonomy of memory errors. We assume the program is not malware. This taxonomy is mainly derived from two systematic survey papers [37, 38].

How to defend a program against memory errors has been extensively studied in recent years. For each category in Table 1, we are able to find corresponding defenses. In this paper, we are particularly interested in the *sanitizer*-style techniques because they thoroughly enforce sanity checks in the program to immediately catch memory errors before they are exploited.

## 2.2 N-version System

The concept of the N-version system was initially introduced as a software fault-tolerance technique [9] and was later applied in enhancing software security [5, 20, 39, 44]. In general, the benefit of the N-version system is that an attacker is required to simultaneously compromise all variants with the same input in order to take down the whole system. To achieve this benefit, an N-version system should have at least two components: 1) a variant producer that generates diversified variants based on predefined principles and 2) an execution engine (NXE) that synchronizes and monitors the execution of all program variants. We differentiate BUNSHIN with related works along these two lines of work.

**Diversification.** Diversification techniques represent the intended protection goal of an N-version system, for example, using complementary scheduling algorithms to survive concurrency errors [39]; using dynamic control-flow diversity and noise injection to thwart cache side-channel attacks [11]; randomizing sensitive data to mitigate privacy leaks [8, 45]; running multiple versions to survive update bugs [20]; using different browser implementations to survive vendor-specific attacks [44]. Diversifica-

tion can also be done in load/run time such as running program variants in disjoint memory layouts to mitigate code reuse attacks [7, 10, 41].

Differently, BUNSHIN aims to reduce the execution slowdown and conflicts of security mechanisms. To achieve this goal, two protection distribution mechanisms are proposed. Partial DA checking [31] also attempted to improve the performance of dynamic analysis with the N-version approach. However, it does not provide any protection distribution mechanism; instead it just insecurely skips checking some syscalls to improve performance. In addition, attacks can be completed by exploiting the vanilla variant before other protected variants find out. In contrast, BUNSHIN proposes two principled diversification techniques to achieve this goal and also presents a more robust NXE with thorough evaluation.

**NXE.** Depending on how the program variants are generated, an NXE is designed to synchronize variants at different levels, such as instruction/function level [39], syscall level [7, 10, 21, 24, 32, 42] or, file/socket IO level [44]. BUNSHIN shares some common features with other syscall-based NXE systems, including syscall divergence comparison (both sequence and arguments), virtual syscall and signal handling, and a leader-follower execution pattern backed by a ring-buffer-based data structure for efficient event streaming [21, 24, 42]. However, BUNSHIN differs from these works in the following ways:

*Syscall hooking.* Prior works hook syscalls using a customized kernel [10], which jeopardizes its deployability; using the Linux `ptrace` mechanism, which causes high synchronization overhead due to multiple context switches per each syscall [7, 10, 32, 42]; or binary-rewriting the program to redirect a syscall to a trampoline [21], which may break the semantics of the program when replacing an instruction with fewer bytes with one with more bytes. MvArmor [24] leverages `Dune` [3] and Intel VT-x. However, it incurs a high overhead for syscalls that needs passthrough and is also subject to the limitations of `Dune` (e.g., signal and threading). To tackle these issues, BUNSHIN hooks syscalls by temporarily patching the syscall table with a loadable kernel module.

*Multithreading support.* Multithreading support varies in proposed NXEs, such as allowing only processes to be forked, not threads [10]; enforcing syscall-ordering across the threads [21, 24, 32], which can easily cause deviations, as not all threading primitives involves syscalls (e.g., `pthread_mutex_lock`); and using CPU page fault exception to synchronize all memory operations [7], which leads to high overhead. None of these works are evaluated on multithreading benchmarks like PARSEC or SPLASH-2x. ReMon [42] seems to have a similar level of support for multithreading as BUNSHIN—race-free programs by injecting synchronization agents into the compiled binary, as discussed in [40]. BUNSHIN borrows the *weak determinism* concept from the deterministic multithreading (DMT) domain and fully describes its design and implementation details to support it. BUNSHIN also identifies its limitations and potential solutions.

### 2.3 Security/performance Trade-offs

Another approach to fit security into the performance budget is to devise a subset of protections from a full-fledged technique. Compared with Softbound [28], which handles both code and data pointers, CPI [25] only instruments code pointers, which are less prevalent but more critical to code-reuse attacks [34]. Since the number of sanity checks inserted is dramatically reduced, CPI reduces the performance overhead from about 70% to 8.4%. Similarly, ASAP [43] keeps less commonly executed (i.e., less costly) checks and removes hot checks. However, selective protections sacrifice security. The assumption that security is proportional to sanity check coverage is not valid in many cases. More specifically, say a program contains two exploitable buffer overflow vulnerabilities; eliminating only one does not actually improve the security, as one bug is enough for the adversary to launch the attack. Unlike CPI and ASAP, BUNSHIN is a novel concept to reduce the slowdown caused by security mechanisms without sacrificing any security.

## 3 Design

In a typical N-version system, program variants are executed in parallel and synchronized by an execution engine to detect any behavior divergence. The whole system terminates only when all variants have terminated. Therefore, the overall runtime of an N-version system can be decomposed into two parts: 1) the time required to execute the slowest variant, and 2) any additional time used for variant synchronization and monitoring.

### 3.1 Protection Distribution Principle

BUNSHIN's protection distribution applies to *sanitizer*-style techniques, which have three properties: 1) They enforce security via instrumenting the program with runtime checks; 2) All the checks instrumented are independent of each other in terms of correctness; and 3) A sanity check alters control flow when and only when the check fails, i.e., the program behaves normally when no memory errors or attacks are present. The majority of memory error prevention techniques are *sanitizer*-style, including stack cookies, CFI, CPI, SAFECode, ASan, MSan, UBSan, Softbound, and CETS, which is also the foundation of profiling-guided security retrofitting such as ASAP [43] and Multicompiler [19].

These properties allow BUNSHIN to 1) measure runtime overhead imposed by the sanitizer as well as remove sanity checks; 2) split the program to allow only portions of the program to be instrumented with sanity checks; 3) split the set of security techniques to allow only selected

checks to be enforced; and 4) produce functionally similar program variants such that BUNSHIN can synchronize their executions and reason about behavior divergences. BUNSHIN distributes checks with two principles.

*Check distribution* takes a single security technique (e.g., ASan) as it is and distributes its runtime checks on a program over N program variants. Specifically, BUNSHIN first splits the program into several disjoint portions and then generates a set of variants, each with only one portion of the program instrumented by the technique. Since only a fraction of the code is instrumented with security checks, the slowdown for each variant is smaller compared with a fully instrumented program. And given that all portions of the target program are covered through the collection of the variants, the security protection is the same as if the security mechanism is applied to the whole program.

In the example of ASan, the splitting unit is every function in the program and for a 3-variant split, after *check distribution*, each variant has ⅓ functions instrumented with sanity checks and the other ⅔ uninstrumented, while collectively, all functions are covered.

It is worth noting that instrumentations added by sanitizers fall into two categories: metadata maintenance (e.g., bound and alias information in the example of ASan or SoftBound) and sanity checks. BUNSHIN does not remove instructions related to metadata maintenance, as removing them will break the correctness of a sanitizer.

*Sanitizer distribution* takes multiple security techniques and distributes them over N variants. Specifically, BUNSHIN first splits these security mechanisms into several disjoint groups whereby each group contains security mechanisms that are collectively enforceable to the program, i.e., they do not conflict with each other. Since only a subset of the intended protections is enforced on each variant, the slowdown for each variant is smaller compared with the case when all protection techniques are enforced on the same program (if ever possible). Another important benefit of *sanitizer distribution* is that by distributing security mechanisms to multiple program variants, any conflicts between them (e.g., ASan and MSan) can be avoided without re-engineering these security mechanisms. Since all intended protections are enforced through the collection of variants, the overall protection is the same as if all mechanisms are applied to the whole program.

In the example of UBSan, the splitting unit is every sub-sanitizer in UBSan, such as `integer-overflow` and `divide-by-zero`. For a 3-variant split, after *check distribution*, each variant has a disjoint set of sub-sanitizers (i.e., `integer-overflow` appears in only one variant), while collectively, all sub-sanitizers are covered.

Due to space constraints, interested readers may find a formal modeling of BUNSHIN's protection distribution principles at http://arxiv.org/abs/1705.09165.

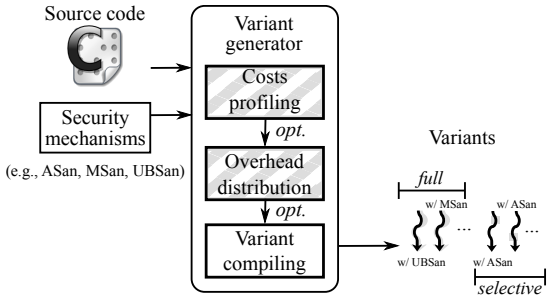


**Figure 1:** Variant generator workflow

### 3.2 Automated Variant Generator

Figure 1 illustrates the high-level workflow of the variant generator. The generator first compiles the target program without any security mechanisms enforced and runs it with the profiling tool to get a baseline profile. In the *check distribution* case, the generator then compiles and profiles the program with the intended security mechanism. In the *sanitizer distribution* case, the generator compiles and profiles the program multiple times, each time with one of the intended security mechanisms. The overhead profile is derived by comparing the security-enforced profiles with the baseline profile. In the next step, the generator runs the overhead distribution algorithm with the intended number of splits (the N value) and creates N build configurations for the compilers, each corresponding to one program variant. The goal is to distribute the overhead measured by the profiling step *fairly* to each variants such that all variants finish execution at approximately the same time. Finally, the generator compiles the program N times, each with a different build configuration, to get N program variants.

**Profiling.** BUNSHIN relies on profiling to obtain the runtime slowdown numbers as the inputs to the overhead distribution algorithm. We choose to explicitly rely on profiling because it is a reliable way to obtain the actual cost of a particular sanity check without making assumptions about the nature of the program or the sanitizer. It also takes in the effect of not only extra CPU cycles required to run the check, but also the side effects on cache-line usage, register pressure, or memory allocations. However, the profiling approach does require an adequate and representative workload to simulate the usage patterns in a production environment. Fortunately, for many projects, such a workload is often available in a form of test suites, which can be directly used to build a profile. More sophisticated profiling tools [2, 15] are orthogonal to BUNSHIN and can be leveraged to improve the overhead profiling if necessary. After profiling, the sanity checks are distributed to N variants in a way that the sum of overhead in each variant is almost the same.

**Variant compiling.** Variant compiling for *check distribution* is essentially a "de-instrumentation" process that involves deleting the instructions that are only used for

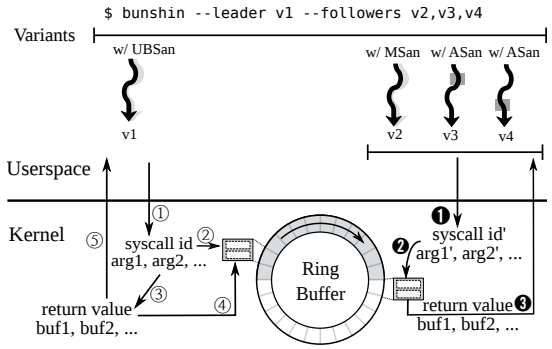


**Figure 2:** General synchronization procedure. The synchronization is triggered when the syscall is trapped into kernel, as denoted by ①❶ in both the leader and follower path. The leader then checks-in the syscall arguments to the shared slot (②), executes the syscall (③), and turns-in the execution results in the shared slot (④). A follower first checks whether the syscall arguments stored in the slot match its own arguments (❷ ) and if they match, directly fetches the results from sync slot (❸ ) without actually performing the syscall. The difference between *lockstep* mode and *ring-buffer* mode lies in whether step ③ for the leader can be performed before step ❷ for all followers is completed.

sanity checks instrumented by sanitizers. In order to collect such instructions for deletion, BUNSHIN uses data and control dependence information maintained during the compilation process and performs *backward slicing* to automatically collect sanity check-related instructions and discard them. Variant compiling for *sanitizer distribution* is trivial, as it can be done by simply compiling the program with the compilation settings a user would normally use for those sanitizers, as long as the sanitizers used to harden the program are collectively enforceable.

### 3.3 N-version Execution Engine

BUNSHIN's NXE synchronizes the executions of N program variants and makes them appear as a single instance to any external entity. We present and justify various design efforts to improve BUNSHIN's NXE in efficiency and robustness.

**Strict- and selective- lockstep.** To synchronize the leader and the follower instances, a lockstep at syscalls is required. BUNSHIN provides two lockstep modes: *strict-lockstep* and *selective-lockstep*. In *strict-lockstep* mode, the leader executes the syscall only if all followers have arrived and agreed on the syscall sequence and arguments. This ensures the security guarantee—the attack cannot complete in either instance. The downside is that variants are frequently scheduled in and out of the CPU due to the necessary waiting, leading to higher runtime slowdown.

We observed that many attacks always trigger certain syscalls before the actual damages are caused. For example, with ASLR enabled, attacks (e.g., ROP) generally leak an address first via I/O write syscalls and then use the leaked address to construct subsequent attack payloads. Based on this observation, BUNSHIN also provides the *selective-lockstep* so that users can choose to prevent the attacks with higher performance. Specifically, BUNSHIN uses the *ring-buffer* mechanism to synchronize instances—the leader executes at near full speed and keeps dumping the syscall arguments and results into the shared ring buffer without waiting, unless the buffer is full. The followers consume the syscall arguments and results at their own speed. Meanwhile, lockstep is still enforced for the selected syscalls (e.g., write related), as illustrated in Figure 2. Our evaluation §5.2, shows that *selective-lockstep* reduces the synchronization overhead by 0.3%-6.3% compared with the *strict-lockstep* mode.

In short, *strict-lockstep* should perfectly preserve the security guarantee of the underlying sanitizer, while *selective-lockstep* is an option we provide when ASLR is enabled, as any remote code-reuse attacks (e.g., return-to-libc and ROP) will have to first leak code/data pointers to bypass ASLR. *Selective-lockstep* is able to stop such attacks by catching the leaks at I/O writes. In fact, any information leak attempt that involves a pointer will be detected at I/O writes. A detailed analysis of the security guarantees provided by BUNSHIN is evaluated in §5.3.

**Multi-threading.** BUNSHIN supports multi-process/thread programs by assigning each group of leader-follower processes to the same *execution group*, and each execution group has its own shared buffers. The starting processes of leader and follower variants form the first execution group, and when the leader forks a child, the child automatically becomes the leader in the new execution group. The child of a follower variant automatically becomes a follower in the new execution group. In fact, for daemon-like programs, (e.g., Apache, Nginx, sshd), simply separating parent and children processes into different execution groups can be sufficient to eliminate syscall sequence variations caused by non-deterministic schedulers because for those programs, each thread/process is highly independent of the others and hardly ever or never updates shared data.

However, for general-purpose multi-thread programs, synchronizing shared memory accesses is necessary to ensure that the leader and followers have consistent views on shared data. This can be achieved by enforcing all followers to follow exactly the same order of shared memory accesses as the leader (*strong determinism*), which can hardly be achieved without a high performance penalty, as evidenced in the deterministic multi-threading (DMT) domain [4, 30]. As a compromise, inspired by Kendo [30], BUNSHIN ensures only that all followers follow exactly the same order of all lock acquisitions as the leader (*weak determinism*). For example, if thread 1 in the leader acquires a mutex before thread 2 passes a barrier, the same order will be enforced in all followers. For programs without data races, strong determinism and weak determinism

offer equivalent guarantees [30]. BUNSHIN achieves this with an additional 8.5% overhead on SPLASH-2x and PARSEC benchmarks (§5.2).

We argue that ensuring *weak determinism* is sufficient for the majority of multi-thread programs, as race-free programming is encouraged and tools have been proposed to help developers eliminate data races [17, 18]. However, should this becomes a problem in the future, BUNSHIN is capable of plugging in sophisticated DMT solutions such as DThreads [26] with minor adjustments.

**Sanitizer-introduced syscalls.** Memory safety techniques generally issue additional syscalls during program execution to facilitate sanity checks. With all sanitizers we tested, i.e., ASan, MSan, UBSan, Softbound, CETS, CPI, and SAFECode, all introduced syscalls can be categorized into three classes: 1) pre-launch data collection, 2) in-execution memory management, and 3) post-exit report generation. To illustrate, before executing the `main` function, ASan goes through a data collection phase by reading various files in `/proc/self` directory (on Linux system). During program execution, ASan issues additional memory-related syscalls for metadata management. Upon program exit, ASan might invoke external programs to generate human readable reports.

Given that variants instrumented with different sanity checks are expected to have diverged syscall sequences, BUNSHIN needs to address this issue to avoid false alerts. In achieving this, BUNSHIN 1) starts synchronization only when a program enters its `main` function; 2) ignores all the memory management-related syscalls; and 3) stops synchronization by registering as the first `exit` handler.

We verified that all the syscall divergences caused by the aforementioned sanitizers are successfully resolved. Although this is only an empirical verification, we believe this can be a general solution because any practical security mechanism should not alter program semantics in normal execution states, which, reflected to the outside entities, are syscall sequences and arguments.

# 4 Implementation

## 4.1 Automated Variant Generator

**Profiling.** To obtain overhead data for *check distribution*, BUNSHIN instruments the program with performance counters based on how the underlying sanitizer works. As a prototype system, BUNSHIN currently measures the execution time of all program functions based on the observation that the majority of memory-related security checks (as discussed in §2.1) operate at function level. We discuss how to perform profiling instrumentation in a generic way in §6. Obtaining profiling data for *sanitizer distribution* is easy, as no extra instrumentation is needed. BUNSHIN runs the program with each security mechanism individually enforced and obtains the overall execution time.

**Check removal.** BUNSHIN removes sanity checks at function level and the process consists of two steps:

In the *discovery* step, BUNSHIN compiles a baseline version and an instrumented version of the same program and then uses an analysis pass to dump the added/modified basic blocks per function. Among these basic blocks, BUNSHIN considers a basic block that 1) is a branch target, 2) contains one of the known sanity check handler functions (e.g., in the case of ASan, functions prefixed with `__asan_report_`), and 3) ends with the special LLVM `unreachable` instruction as a *sink* point for security checks. This is based on the properties of sanitizers, as a sanity check should preserve program semantics, i.e., special procedures are only invoked when a sanity check fails. Instrumentations for metadata maintenance involves neither sanity check functions nor `unreachable` instructions and hence are filtered out in this step.

In the *removal* step, BUNSHIN automatically reconstructs sanity checks based on the observation that sanity checks are instructions that branch to the *sink* points found in *discovery* step. After identifying the branching points and the corresponding condition variables, BUNSHIN performs a recursive backward trace to variables and instructions that lead to the derivation of the condition variable and marks these instructions during tracing. The backward trace stops when it encounters a variable that is not only used in deriving the value of the condition variable but also used elsewhere in the program, an indication that it does not belong to the sanity check. Removing the sanity check is achieved by removing all marked instructions found in the above process. This functionality is implemented as an LLVM pass.

## 4.2 N-version Execution Engine

**Pthreads locking primitives.** BUNSHIN enforces *weak determinism* discussed in §3.3 by re-implementing the full range of synchronization operations supported by `pthreads` API, including locks, condition variables, and barriers. BUNSHIN introduces a new syscall, `synccall`, specifically for this purpose. `synccall` is exposed to processes under synchronization by hooking an unimplemented syscall in an x86-64 Linux kernel (`tuxcall`). In the kernel module, BUNSHIN maintains an `order_list` to record the total ordering of locking primitive executions. When a leader thread hits a primitive, e.g., `pthread_mutex_lock`, it calls `synccall` to atomically put its *execution group* id (EGID) in the `order_list` and wake up any follower threads waiting on its EGID before executing the primitive. When a follower thread hits a primitive, the call to `synccall` will first check whether it is the thread's turn to proceed by comparing its EGID and the next EGID in the `order_list`. The thread will proceed if it matches; otherwise, it puts itself into a variant-specific waitqueue. If the primitive may cause the thread to sleep

(such as a mutex), the thread wakes up its next sibling in the waitqueue, if there are any, before sleeping.

Hooking `pthreads`' locking primitives is done by placing the patched primitives in a shared library, which is guaranteed to be loaded earlier than `libpthread`.

The drawbacks of this implementation are also obvious: 1) BUNSHIN is unable to handle multi-thread programs that are built with other threading libraries or that use non-standard synchronization primitives (e.g., using `futex` directly); 2) The performance overhead of BUNSHIN increases linearly with the usage frequency of these primitive operations. Fortunately, the majority of multi-thread programs are compatible with `pthreads` and locking primitives are only used to guard critical sections, which represent only a small fraction of execution.

**Shared memory access.** Similar to the approach used by MSan to trace uninitialized memory accesses, whenever BUNSHIN detects mapping of shared memory into the variant's address space (by the indication of `mmap` syscall with specific flag combinations), it creates shadow memory copies of the same size and then marks them as "poisoned" state `HWPOISON` whereby any access attempts to the mapped memory will also lead to an access attempt to the shadow copy, which eventually triggers a signal (`SIGBUS`). Upon capturing the signal, memory access is synchronized in the normal way syscalls are handled, i.e., compare and copy content of the accessed address from the leader's mapping to the followers mapping.

**Workflow.** BUNSHIN can be started with the path to each variant and the program arguments. BUNSHIN first informs the kernel module to patch the syscall table and then sets the `LD_PRELOAD` environment variable to the library containing patched virtual syscalls and pthread locking primitives. It then forks N times and launches one program variant in each child process. After that, BUNSHIN pauses and waits for status change of the variants. If any of the variant process is killed due to behavior divergences, BUNSHIN alerts and aborts all variants. Otherwise, it exits when all variants terminate.

# 5 Evaluation

In this section, we first evaluate BUNSHIN's NXE in terms of robustness and efficiency. In particular, we run BUNSHIN on various programs and empirically show that BUNSHIN is capable of handling the majority of them with low overhead and no false alarms. We also empirically test whether BUNSHIN can provide the same level of security guarantee as the underlying sanitizers, in other words, whether BUNSHIN might compromise the security by partitioning the program or splitting the sanitizers.

We then showcase how to accelerate ASan-hardened programs with *check distribution* and UBSan-hardened programs with *sanitizer distribution*. We use another case study – combining ASan, MSan, and UBSan – to show that BUNSHIN is capable of unifying security mechanisms that have conflicted implementations.

We also evaluate BUNSHIN in terms of hardware resource consumption, which could limit BUNSHIN's applicability, and report the performance of BUNSHIN under various levels of system load.

**Experiment setup.** The experiments are primarily conducted on a machine with Intel Xeon E5-1620 CPU (4 cores) and 64GB RAM running 64-bit Ubuntu 14.04 LTS, except the experiment on scalability, which is done with Intel Xeon E5-2658 (12 cores), and the experiment on RIPE benchmark, which is done on a 32-bit virtual machine. For evaluations on web servers, we dedicate another machine to launch requests and measure server response time. The client machine is connected to the experiment machine with a direct network cable. The associated network card permits 1000Mb/s bandwidth. Unless stated otherwise, the NXE is configured to run in *strict-lockstep* mode for stronger security guarantee.

## 5.1 NXE Robustness

We use a mixed sample of CPU-intensive and IO-intensive programs for experiments, including SPEC2006 benchmark representing single-thread programs, PARSEC, and SPLASH-2x benchmark for multi-threaded programs, and Nginx and Lighttpd as representative server programs. For each sample program, we compile it with the LLVM compiler framework and run the same binary on BUNSHIN's NXE, i.e., BUNSHIN will synchronize identical N binaries. This is to (empirically) verify the robustness of BUNSHIN's NXE design.

We do not observe false positives in any experiments on SPEC, SPLASH-2x, Nginx, and Lighttpd. However, BUNSHIN is only able to run on six out of 13 programs in the PARSEC benchmark. `raytrace` would not build under `clang` with `-flto` enabled. `canneal`, `facesim`, `ferret`, and `x264` intentionally allow for data races. `fluidanimate` uses ad-hoc synchronization and hence, bypassing `pthreads` APIs and `freqmine` does not use `pthreads` for threading. These represent the limitation of BUNSHIN's NXE: enforcing only weak determinism on `pthreads` APIs.

## 5.2 NXE Efficiency & Scalability

Figure 3 and Figure 4 show the efficiency evaluation of the NXE under both strict- and selective-lockstep modes when synchronizing 3 variants. For the SPEC2006 benchmark, the average slowdowns for the two modes are 8.1% and 5.3%, respectively. The overhead is sightly higher on multi-threaded benchmarks (SPLASH-2x and PARSEC) – 15.7% and 13.8%. This is due to the additional overhead for recording and enforcing the total order of locking primitive acquisitions. The selective-lockstep mode reduces the overhead by 0.3%-6.3% in the benchmark programs.

We further evaluate the efficiency of BUNSHIN's NXE

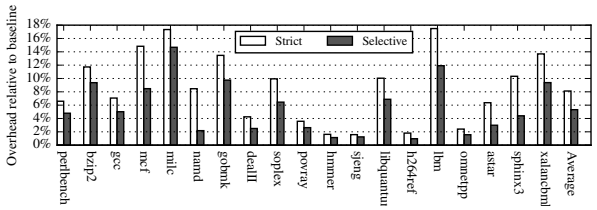


**Figure 3:** Evaluation of BUNSHIN' NXE efficiency with SPEC2006.

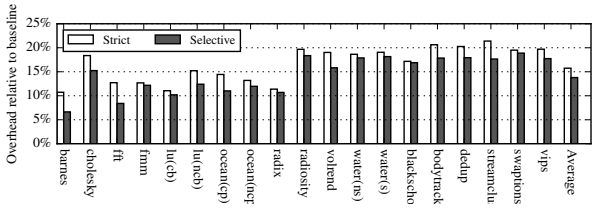


**Figure 4:** Evaluation of BUNSHIN's NXE efficiency with SPLASH-2x and PARSEC (number of threads = 4).

on two server programs, `lighttpd`, representing single-thread servers, and `nginx`, representing multi-threaded servers. We synchronize 3 variants and for `nginx`, we run 4 worker threads, the default value after installation. We simulate various workload situations by using 64 (light), 512 (heavy), and 1024 (saturated) concurrent connections and simulate HTTP requests to files of 1KB and 1MB.

The results are shown in Table 2. A noticeable difference is that the percentage overhead when requesting small files (e.g., 1KB), is significantly larger compared with requesting large files (e.g., 1MB). The reason is that, while the absolute value of overhead is comparable in both situations, it can be better amortized into the networking time of a large file, therefore leading to smaller relative overhead. We believe that in real-world settings when the servers are connected to LANs and WANs, even the overhead for smaller files can be amortized in the networking time, leading to unnoticeable overhead.

Figure 5 shows the scalability of BUNSHIN's NXE in terms of total number of variants synchronized. We use a 12-core machine for this experiment as the number of variants should not exceed the number of cores available. As the number of variants goes from 2 to 8, the overhead increases from 0.9% to 21% accordingly. The primary reason for overhead increase is the LLC cache pressure, as the cache miss rates increase exponentially when more variants are executed in parallel. Recently added CPU features such as Intel Cache Allocation Technology [22] might help to mitigate this problem.

### 5.3 Security Guarantee

BUNSHIN does not remove any sanity checks, but only distributes them into multiple variants. In *strict-lockstep* mode, BUNSHIN should not compromise the intended security guarantee, as no variant can proceed with a syscall without the arrival of other variants. Conceptually, the only way to compromise all variants is to launch an attack that is out of the protection scope of the underlying

| Config | Conn | Base | Strict | | Selective | |
|---|---|---|---|---|---|---|
| `lighttpd` | 64 | 10.3 | 11.9 | 15.3% | 11.8 | 14.6% |
| 1 Process | 512 | 8.71 | 10.5 | 20.5% | 10.1 | 15.7% |
| 1K File | 1024 | 8.48 | 10.4 | 22.6% | 10.1 | 19.3% |
| `lighttpd` | 64 | 974 | 994 | 2.05% | 992 | 1.85% |
| 1 Process | 512 | 959 | 972 | 1.35% | 970 | 1.15% |
| 1M File | 1024 | 955 | 964 | 0.91% | 961 | 0.63% |
| `nginx` | 64 | 9.81 | 11.6 | 18.7% | 11.2 | 14.3% |
| 4 Threads | 512 | 8.46 | 10.3 | 21.9% | 9.88 | 16.8% |
| 1K File | 1024 | 8.20 | 10.2 | 24.4% | 9.63 | 17.4% |
| `nginx` | 64 | 950 | 967 | 1.79% | 964 | 1.47% |
| 4 Threads | 512 | 985 | 999 | 1.40% | 996 | 1.12% |
| 1M File | 1024 | 979 | 998 | 1.94% | 995 | 1.63% |
| Ave. (1KB) | | | | 20.56% | | 16.4% |
| Ave. (1MB) | | | | 1.57% | | 1.31% |

**Table 2:** Performance of `lighttpd` and `nginx` under BUNSHIN's NXE, Performance measured as the processing time per request (unit. $\mu$s). We use `apachebench` as test driver and run each experiment 1000 times to reduce the effect of network noise.

sanitizer. Given that there is no attack window between variants, if an input sequentially compromises all variants without causing a divergence, it means the input bypasses all sanity checks. In this case, the attack will also succeed even if all checks are enforced in one variant (i.e., no BUNSHIN). In other words, the attack is essentially not in the protection scope of the underlying sanitizers and hence will not be in the scope of BUNSHIN.

On the other hand, the *selective-lockstep* mode might introduce an attack window between the variants that allow an attacker to potentially compromise them one by one. However, BUNSHIN remains effective if the window is small enough. To quantify the attack window, we measure the syscall distance between the leader and the slowest follower during our experiments. For CPU-intensive programs (SPEC2006, PARSEC, and SPLASH-2x), the average number of syscall gap is 5 while for IO-intensive programs (lighttpd and nginx), the average number of syscall gap is only 1. The gap is small because even in *selective-lockstep* mode, the variants are still strictly synchronized at IO-related syscalls. We believe that this is a small enough time frame to thwart attackers.

To empirically confirm that real-world attacks can be thwarted even in *selective-lockstep* mode, we first evaluated BUNSHIN on the RIPE benchmark with *check distribution* on ASan. In particular, in compiling the programs generated by the RIPE benchmark, we go through the normal *check distribution* procedure to produce two variants, and then launch and synchronize them with our NXE. The results in Table 3 confirm that BUNSHIN does not compromise the intended security guarantee of ASan.

To further verify this, we applied BUNSHIN to five real-world programs, `nginx`, `cpython`, `php`, `openssl`, and `httpd`, which contain known vulnerabilities that can be detected by ASan (to evaluate *check distribution*) and UBSan (to evaluate *sanitizer distribution*). Similar to the RIPE benchmark case, we apply BUNSHIN on these vulnerable programs to produce two variants and later

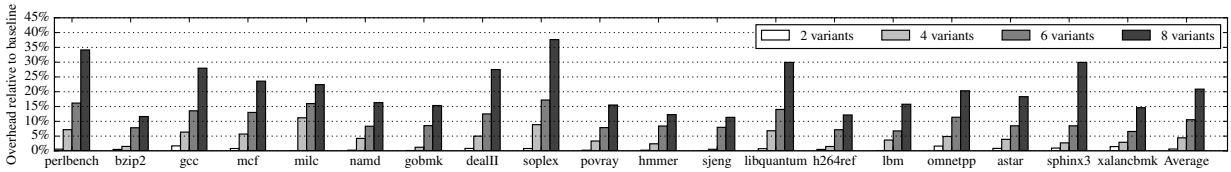


**Figure 5:** Scalability of BUNSHIN in terms of synchronizing 2 to 8 variants. For each program, we show the synchronization overhead over the baseline execution. On average, the overhead almost doubled with 2 more variants synchronized. Different programs show slightly different patterns in overhead growth. One of the reasons could be their differences in cache sensitivity [23].

| Config | Succeed | Probabilistic | Failed | Not possible |
|---|---|---|---|---|
| Default | 114 | 16 | 720 | 2990 |
| ASan | 8 | 0 | 842 | 2990 |
| BUNSHIN | 8 | 0 | 842 | 2990 |

**Table 3:** We first run the RIPE benchmark on vanilla 32-bit Ubuntu 14.04 OS, and 114 exploits always succeed and 16 succeed probabilistically. After adding ASan in the compilation, only 8 exploits succeed. After applying *check distribution* on the programs, still the same 8 exploits succeed.

| Program | CVE | Exploits | Sanitizer | Detect |
|---|---|---|---|---|
| nginx-1.4.0 | 2013-2028 | blind ROP | ASan | Yes |
| cpython-2.7.10 | 2016-5636 | int. overflow | ASan | Yes |
| php-5.6.6 | 2015-4602 | type confusion | ASan | Yes |
| openssl-1.0.1a | 2014-0160 | heartbleed | ASan | Yes |
| httpd-2.4.10 | 2014-3581 | null deref. | UBSan | Yes |

**Table 4:** Empirically test BUNSHIN's security guarantee with real-world programs and CVEs.

launch and synchronize them with our NXE. We then use the same exploit that triggered the warnings from ASan or UBSan to drive the program under BUNSHIN and check to see whether the same warnings are raised. The result show that all exploitation attempts are detected (Table 4).

For a concrete example, we applied BUNSHIN to `nginx-1.4.0`, which contains bug CVE-2013-2028 that can be detected by ASan. We use *check distribution* to produce two variants, A and B and use three published exploits [6, 12, 13] to test whether they can succeed in exploiting the vulnerable `nginx` protected under BUNSHIN. The result shows that, when the overflow is triggered, variant A issues a `write` syscall (trying to write to `stderr`) due to ASan's reporting, while B does not. A further investigation on the protection distribution report shows that the vulnerable function `ngx_http_parse_chunked` is instructed to be protected by variant A, which explains why variant A issues the `write` syscall.

**Attacking BUNSHIN.** Given the attack window in *selective-lockstep* mode, an attacker might be able to complete some simple attacks before detection, such as killing child threads/processes, closing file/sockets, or exhausting resources by allocating large chunks of memory, etc., provided that the attacker can inject shellcode or reuse program code to invoke the call and place the arguments of syscall in correct addresses. An attacker might also launch denial-of-service attacks by sending compromised variants into infinite loops that do not involve synchronize syscalls (in both modes) or sleep/pause indefinitely (in *selective-lockstep* mode).

Another attack vector is BUNSHIN's variant monitor. For example, an attacker might intentionally crash BUNSHIN with unhandled non-deterministic sources such as uninitialized data (e.g., some encryption libraries intentionally use uninitialized data as a source of entropy, although such a practice is discouraged). In addition, although we take care to keep the variant monitor simple and secure, it is not guaranteed to be bug-free. Therefore, if an attacker compromises the variant monitor, he/she might be able to circumvent syscall synchronization.

### 5.4 *Check Distribution* on ASan

We show the effectiveness of *check distribution* in accelerating the performance of programs instrumented with ASan. The reason we choose ASan for the case study is twofold: 1) ASan is representative of how memory error detection techniques are generally enforced – introducing runtime sanity checks. In addition, the majority of checks placed in the program are independent of each other and hence satisfy the assumption of *check distribution*. 2) ASan provides a relatively high coverage on memory safety and hence is appealing for long-living processes (like server programs) to thwart attackers at runtime. However, the slowdown by enforcing ASan to the whole program is the main obstacle in making it useful in production. We hope this experiment will provide insights on how to use ASan through BUNSHIN.

The case study is done with the SPEC2006 benchmark programs using the `train` dataset for profiling and `reference` dataset for the actual performance measurement. On average, the runtime slowdown caused by ASan is reduced from 107% (enforced to the whole program) to 65.6% (2 variants) and 47.1% (3 variants), respectively, about 11% more than ½ and ⅓ of the original slowdown. Due to space constraints, we show only results for the more complex case (3 variants) in Figure 6.

However, we also observed two outliers that do not show overhead distribution: `hmmer` and `lbm`. After investigating their execution profile, we observe that there is a single function that accounts for over 95% of the execution time and the slowdown caused by ASan. Since BUNSHIN performs sanity check distribution at the function level, the overhead is inevitably distributed to one variant, causing that variant to be the bottleneck of the entire system. However, concentrating functionalities in one

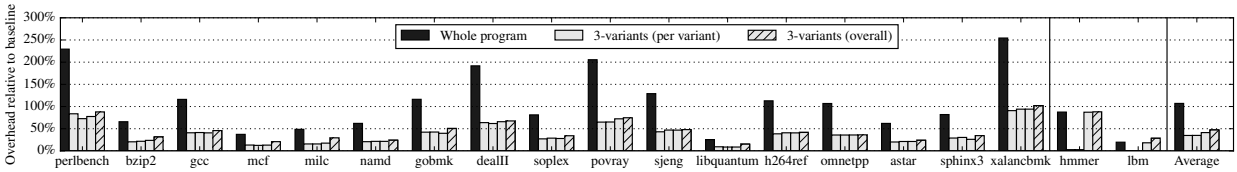


**Figure 6:** Effectiveness of *check distribution* on ASan with three variants. For each program, we show the total overhead if ASan is applied to the whole program as well as per-variant overhead and BUNSHIN overall overhead. The two programs on the right are outliers that do not show overhead distribution.

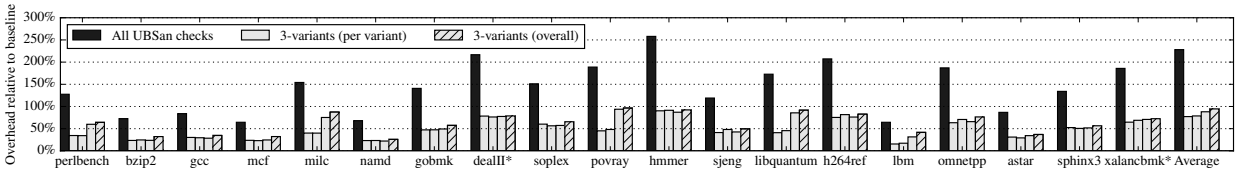


**Figure 7:** Effectiveness of *sanitizer distribution* on UBSan with three variants. For each program, we show the total overhead if all checks of UBSan are enforced as well as per-variant overhead and BUNSHIN overall overhead. For `dealII` and `xalancbmk`, the overhead number is 4x larger than what is shown in the figure.

single function is rarely seen in the real-world software we tested, including Python, Perl, PHP Apache httpd, OrzHttpd, and OpenSSL; hence, we do not believe these outliers impair the practicality of BUNSHIN.

### 5.5 *Sanitizer Distribution* on UBSan

UBSan is a representative example to illustrate why collectively enforcing lightweight sanity checks might lead to significant overall slowdown. UBSan contains 19 sub-sanitizers, each with overhead no more than 40%. However, adding them leads to over 228% overhead on SPEC2006 benchmarks, making UBSan a perfect example to exercise *sanitizer distribution*.

Similar to the ASan case study, the test case is done with SPEC2006 programs using `train` dataset for profiling and `reference` dataset for experimentation. On average, the runtime caused by UBSan is reduced from 228% (enforced all checks) to 129% (2 variants) and 94.5% (3 variants), respectively, about 15% more than ½ and ⅓ of the original slowdown. Due to space constraints, we show only the results for the more complex case (3 variants) in Figure 7. This deviation from the theoretical optimum is a bit larger compared with the ASan case study because we only have 19 elements in the set and hence are less likely to get balanced partitions across variants. However, it still shows the effectiveness of *sanitizer distribution* in accelerating the overall performance.

### 5.6 Unifying LLVM Sanitizers

In theory, BUNSHIN is capable of unifying any security mechanism that falls in the *sanitizers* definition in §3.1. The reason we choose LLVM sanitizers (ASan, MSan, UBSan) for the case study is mainly because: collectively, they provide almost full protection against memory error, which we have not seen in any other work. Unifying them through BUNSHIN might give some insight on how to achieve full memory error protection without any re-engineering effort to these sanitizers.

In this case study, each variant is simply the program compiled with one of the sanitizers with the default compilation settings. We measure the execution time of each program variant when running by itself and also the total execution time of BUNSHIN. The result is reported in Figure 8. On average, the total slowdown of combining these sanitizers is 278%, with only 4.99% more compared with merely enforcing the slowest sanitizer among the three. In other words, paying a little slowdown for synchronization helps bring additional protection provided by the other two sanitizers.

### 5.7 Hardware Resource Consumption

**Memory.** Since all variants are loaded into memory for parallel execution, the basic memory usage is almost linear to the number of variants. This is an inherent trade-off for execution time. In addition, whether *check distribution* helps to split memory overhead caused by a sanitizer depends on the sanitizer' internal working, In the case study of ASan, although each variant executes only a portion of the sanity checks, it still needs to shadow the whole memory space as required by ASan. Therefore, the memory overhead of ASan still applies to each variant. However, the memory overhead can be distributed for shadow stack-based techniques. By definition, *sanitizer distribution* can be used to distribute memory overhead to multiple variants. In the UBSan case study, the memory overhead of each variant is the sum of all enforced sub-sanitizers' overhead.

**CPU cycles.** BUNSHIN's NXE utilizes spare cycles in a multi-core CPU for efficient variant synchronization. If the CPU does not provide sufficient parallelism, BUNSHIN will not be able to improve the performance; instead, it will only introduce more performance overhead. An evaluation on C/C++ programs in the SPEC2006 benchmark shows that the average synchronization overhead is 103.1% when running BUNSHIN on a single core.

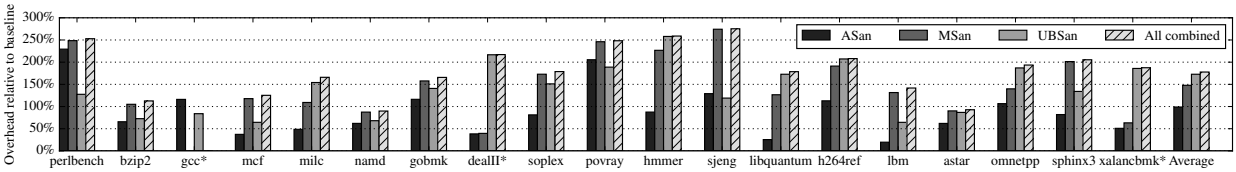


**Figure 8:** Performance result of each LLVM sanitizer respectively as well as the overall performance overhead when unified under BUNSHIN. `gcc` cannot run with MSan, therefore, we exclude the evaluation on it. For `dealII` and `xalancbmk` the overhead number is 4x larger than what is shown in the figure.

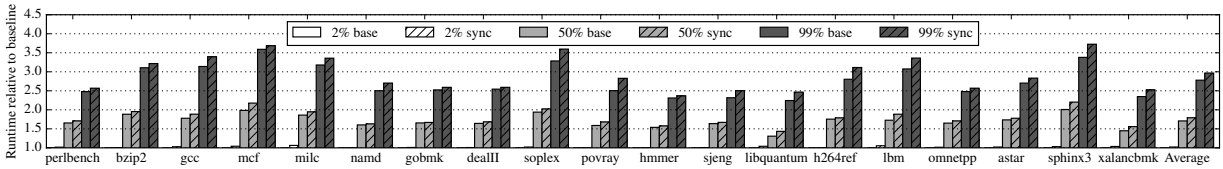


**Figure 9:** Evaluation of BUNSHIN execution engine under various workload levels. The experiment is done with the configuration of synchronizing 2 variants. We use the system-stressing tool `stress-ng` to add background workloads including CPU tasks, cache thrashing, and memory allocations and deallocations. We maintain the background load level at 50% and 99%, respectively. The 2% load for the baseline case is due to the kernel and OS background services.

Although BUNSHIN is not suitable for devices with a single core, it does not mean that BUNSHIN requires exclusive cores to work. In fact, due to OS-level task scheduling, BUNSHIN can exploit free cycles in the CPU as long as not all cores are fully utilized. Figure 9 shows that BUNSHIN's performance is stable under various load levels. The average slowdown due to synchronization is 10.23% and 13.46%, respectively, when the CPU is half and fully loaded, slightly higher than the case when the load is small (8.1%). The results prove that the performance of BUNSHIN is stable across various load levels.

## 6 Discussion

**Trading-off resources for time.** There is no doubt that BUNSHIN's parallelism consumes more hardware resources; hence BUNSHIN is not suitable for cases where hardware resources are scarce. In fact, BUNSHIN's design is inspired by the popularity of multi-core processors and large-size cache and memory, and trade-off resource usages for execution time. BUNSHIN empowers users to make use of available hardware resources to improve both security and runtime performance and sheds lights on how to solve a difficult problem —speeding up hardened programs without sacrificing security —with simply more hardware resources, which are easy to obtain.

**Sanitizer integration.** BUNSHIN currently has no integration with the sanitizers, i.e., it does not require detailed knowledge of how a sanitizer works in order to "de-instrument" the sanity checks. Although this gives BUNSHIN great flexibility, it also prevents BUNSHIN from further optimization. For example, ASan still shadows the whole memory space even when only a subset of sanity checks is performed per program variant, thus leading to increased memory usage in every variant. To solve this, we could modify ASan's logic in a way such that only a portion of the memory space is shadowed in each variant; in other words, we can distribute the memory overhead to all program variants as well.

**Finer-grained sanity check distribution.** As shown in the case of `hmmer` and `lbm`, sanity check distribution at the function level might be too coarse grained if one or a few functions dominate the total execution. Therefore, to enable finer-grained overhead distribution, we plan to look into performing both profiling and check removal at the basic block level.

## 7 Conclusion

We presented BUNSHIN, an N-version system that seamlessly unifies different and even conflicting protection techniques while at same time reducing execution slowdown. BUNSHIN achieves this with two automated variant generation strategies (*check distribution* and *sanitizer distribution*) for distributing security checks to variants and an efficient parallel execution engine that synchronizes and monitors the behaviors of these variants. Our experiment results show that BUNSHIN is a practical system that can significantly reduce slowdown of sanitizers (e.g., 107% to 47.1% for ASan, 228% to 94.5% for UBSan) and collectively enforce ASan, MSan, and UBSan without conflicts with only 4.99% incremental overhead.

## 8 Acknowledgment

We thank our shepherd, Ittay Eyal, and the anonymous reviewers for their helpful feedback. This research was supported by NSF under award DGE-1500084, CNS-1563848, CRI-1629851, CNS-1017265, CNS-0831300, and CNS-1149051, ONR under grant N000140911042 and N000141512162, DHS under contract No. N66001-12-C-0133, United States Air Force under contract No. FA8650-10-C-7025, DARPA under contract No. DARPA FA8650-15-C-7556, and DARPA HR0011-16-C-0059, and ETRI MSIP/IITP[B0101-15-0644].

## References


[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, November 2005.

[2] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Snowbird, Utah, June 2001.

[3] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazieres, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, October 2012.

[4] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: A compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Pittsburgh, PA, March 2010.

[5] Philippe Bergheaud, Dinesh Subhraveti, and Marc Vertes. Fault tolerance in multiprocessor systems via application cloning. In *Proceedings of the 27th IEEE International Conference on Distributed Computing Systems (ICDCS)*, Toronto, Canada, June 2007.

[6] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. Blind return oriented programming (brop), 2014. http://www.scs.stanford.edu/brop.

[7] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. DiversifiedÃ process replicÃę for defeating memory error exploits. In *Proceedings of the 2007 International Performance, Computing, and Communications Conference (IPCCC)*, New Orleans, LA, April 2007.

[8] Roberto Capizzi, Antonio Longo, V. N. Venkatakrishnan, and A. Prasad Sistla. Preventing information leaks through shadow executions. In *Proceedings of the 2008 International Conference on Software Engineering (ICSE)*, Anaheim, CA, December 2008.

[9] Liming Chen and Algirdas Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Fault-Tolerant Computing, 1995*, Jun. 1995.

[10] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *Proceedings of the 15th Usenix Security Symposium (Security)*, Vancouver, Canada, July 2006.

[11] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[12] Exploit Database. nginx 1.3.9-1.4.0 - dos poc, 2013. https://www.exploit-db.com/exploits/25499.

[13] Exploit Database. nginx 1.3.9/1.4.0 x86 - brute force exploit, 2013. https://www.exploit-db.com/exploits/26737.

[14] CVE Details. Vulnerabilities By Date, 2016. http://www.cvedetails.com/browse-by-date.php.

[15] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: less is more. *ACM SIGARCH Computer Architecture News*, December 2000.

[16] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, 2014.

[17] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. Ifrit: interference-free regions for dynamic data-race detection. In *Proceedings of the 23th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Tucson, AZ, October 2012.

[18] Cormac Flanagan and Stephen N Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (PoPL)*, Venice, Italy, January 2004.

[19] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. Profile-guided Automated Software Diversity. In *Proceedings of the 2013 International Symposium on Code Generation and Optimization (CGO)*, Shenzhen, China, February 2013.

[20] Petr Hosek and Cristian Cadar. Safe software updates via multi-version execution. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, May 2013.

[21] Petr Hosek and Cristian Cadar. Varan the unbelievable, an efficient n-version execution framework. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Istanbul, Turkey, March 2015.

[22] Intel. Improving real-time performance by utilizing cache allocation technology, 2015. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf.

[23] Aamer Jaleel. Memory characterization of workloads using instrumentation-driven simulation, 2010. http://www.jaleels.org/ajaleel/publications/SPECanalysis.pdf.

[24] Koen Koning, Herbert Bos, and Cristiano Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *Proceedings of the 46th International Conference on Dependable Systems and Networks (DSN)*, Toulouse, France, June 2016.

[25] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Down Song. Code pointer integrity. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, October 2014.

[26] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient and deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, October 2011.

[27] LLVM. UndefinedBehaviorSanitizer (UBSan) is a fast undefined behavior detector, Feb. 2015. http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[28] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, June 2009.

[29] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler enforced temporal safety for C. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM)*, Toronto, Canada, June 2010.

[30] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Washington, DC, March 2009.

[31] Luis Pina and Cristian Cadar. Towards deployment-time dynamic analysis of server applications. In *Proceedings of the 13th Inter-*

*national Workshop on Dynamic Analysis (WODA)*, Pittsburgh, PA, October 2015.

[32] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the ACM EuroSys Conference*, Nuremberg, Germany, March 2009.

[33] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 ATC Annual Technical Conference (ATC)*, Boston, MA, June 2012.

[34] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, October–November 2007.

[35] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, October 2004.

[36] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 2015 International Symposium on Code Generation and Optimization (CGO)*, San Francisco, CA, February 2015.

[37] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[38] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: The past, the present, and the future. In *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, Amsterdam, Netherlands, September 2012.

[39] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Detecting and surviving data races using complementary schedules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, October 2011.

[40] Stijn Volckaert, Bart Coppens, Bjorn De Sutter, Koen De Bosschere, Per Larsen, and Michael Franz. Taming Parallelism in a Multi-Variant Execution Environment. In *Proceedings of the ACM EuroSys Conference*, Belgrade, Serbia, April 2017.

[41] Stijn Volckaert, Bart Coppens, and Bjorn De Sutte. Cloning your gadgets: Complete ROP attack immunity with multi-variant execution. *IEEE Transactions on Dependable and Secure Computing*, 13(4):437–450, July 2016.

[42] Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. Secure and efficient application monitoring and replication. In *Proceedings of the 2016 ATC Annual Technical Conference (ATC)*, Denver, CO, June 2016.

[43] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High system-code security with low overhead. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[44] Hui Xue, Nathan Dautenhahn, and Samuel T. King. Using replicated execution for a more secure and reliable web browser. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2012.

[45] Aydan Yumerefendi, Benjamin Mickle, and Landon P. Cox. Tightlip: Keeping applications from spilling the beans. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, Cambridge, MA, April 2007.