DART: A Real-Time Address-Randomization Defense with Predictable Timing

Patrick Dobranowski
Vanderbilt University, Nashville, TN, USA

Owen Rice
Vanderbilt University, Nashville, TN, USA

Ryan Burrow
MIT Lincoln Laboratory, Lexington, MA, USA

Nathan Burow
MIT Lincoln Laboratory, Lexington, MA, USA

Bryan C. Ward
Vanderbilt University, Nashville, TN, USA

Abstract

Embedded and real-time systems are increasingly connected and deployed in safety and mission-critical environments, making them a persistent target for attacks capable of compromising industrial control systems and other embedded devices. At the same time, these devices often have strict real-time requirements that require predictable worst-case performance. However, many strong and widely deployed software-security defenses are designed and evaluated with respect to average-case performance, a more important metric in enterprise systems. The worst-case performance of such defenses is not well understood and indeed such defenses are less commonly deployed in embedded systems. In particular, one class of commonly deployed defenses in enterprise systems is code randomization, which protects a system by altering the layout of the virtual address space so that attackers cannot easily target specific parts of a vulnerable application, but randomization is often seen as fundamentally counter to real-time predictability.

This paper presents DART, a real-time address randomization defense with page-level randomization. DART randomizes code in the virtual address space at page-level granularity under placement constraints that move cache behavior from a runtime OS-allocator property to a statically encoded binary property, allowing for timing analysis. An analysis of DART's timing behavior on a real-time testbed demonstrates how the design makes layout-induced timing variance bounded and characterizable across the space of layouts produced, supporting predictable execution-time analysis. The resulting layout search space is then analyzed, and a closed-form expression for the randomization entropy induced by DART is derived. Evaluation results across TACLeBench binaries show increased combinatorial entropy with modest numbers of virtual memory pages per cache color, providing a suitable defense that outperforms traditional virtual-memory protections for attacks such as partial-pointer overwriting or more broadly control-flow hijacking.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Security and privacy → Systems security; Computer systems organization → Embedded and cyber-physical systems; Software and its engineering → Real-time systems software

Keywords and phrases real-time systems, address-space layout randomization, code randomization, worst-case execution time, cache coloring, embedded systems security

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2026.10

Supplementary Material *Software (ECRTS 2026 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.12.2.8`

© Patrick Dobranowski, Owen Rice, Ryan Burrow, Nathan Burow, and Bryan C. Ward;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

Embedded and real-time systems are increasingly being incorporated in myriad application domains ranging from smart home devices like light switches and routers to critical infrastructure and military mission systems. The trend towards increasing connectivity of such devices, *i.e.,* the Internet of Things (IoT), promises to enable new capabilities and greater efficiencies. However, the network connectivity so essential to such functionality also exposes systems to the potential for remote attack, which may not have been possible previously with either analogue or local digital control. Consequently, cyber defenses are now needed for previously undefended real-time and embedded systems [56].

In recent years, however, attacks on real-time and embedded systems (RTES) have become increasingly prevalent, as attackers have targeted systems such as IoT devices (*e.g.,* Mirai botnet [3]), automotive applications [38], industrial control systems (ICS) (*e.g.,* TRITON attack [17]), and heart monitors have failed from security countermeasures (*i.e.,* virus scan) [22]. The United States Department of Homeland Security issued an advisory for a collection of vulnerabilities called "BadAlloc" reported to affect over 25 different real-time operating systems (RTOSes) used in commercial applications ranging from industrial control to IoT to medical devices [50]. There have also been additional attacks in recent wars, such as the commercial Viasat satellite-based communication system used in Ukraine [51].

A prolific attack type for remote attackers is *control-flow hijacking* [45], in which an attacker is able to divert control of a program to gain the ability to execute arbitrary malicious code on the victim host. Modern systems apply data-execution prevention (DEP) [37], which prevents data, which could potentially be provided by an attacker, from being executed. With this constraint, attackers must *reuse* code. Therefore, attackers hijack control flow by exploiting a memory-corruption vulnerability, which allows them to overwrite a part of memory, to overwrite a code pointer, such as a return address, such that when that pointer is taken it redirects control to an attacker-chosen target in the code of the victim process. This is a well-known threat, yet new memory-corruption vulnerabilities in prolific software are identified every day, and both attack and defense techniques have perpetually evolved to become increasingly sophisticated [48]. This attack technique therefore continues to persist in many software systems today, embedded and real-time systems included.

One class of defenses that combat control-flow hijacking attacks is *code randomization*. Code randomization does not eliminate vulnerabilities in software, but instead renders it more difficult to carry out attacks by moving code such that attackers cannot predict where targeted code is placed. An implementation of code randomization, known as Address-Space Layout Randomization (ASLR) [49] is ubiquitously deployed in general-purpose systems, and is enabled by default in instances of Windows, Mac OS X, and Linux. In ASLR, whole code sections (*e.g.,* `.text`, and shared libraries) are moved in the address space as contiguous sections. However, if an attacker is able to *leak* information about the address space of a victim process, they can infer the locations of other nearby code within that section. For example, if an attacker leaks an address of a `libc` function, they could infer the address of other functions in `libc`. To address this threat of memory leakage, randomization-based defenses have been developed that randomize at increasingly fine granularities including function- [14], basic-block- [29], and instruction-level randomization [25, 27].

However, RTES are designed around worst-case performance instead of the average case. This is especially true for those with more stringent timing requirements, which are common in safety-critical systems. In these domains, deterministic execution patterns facilitate worst-case timing and schedulability analysis. Thus, randomization is seemingly antithetical to

predictable real-time systems as changing the code layout creates non-deterministic execution patterns, which invalidates timing analysis used to prove that systems meet their deadlines. This motivates the need for a randomization-based defense that maintains the security benefits of randomization while keeping layout-induced timing variability sufficiently bounded across the space of layouts that worst-case timing analysis remains tractable.

Many randomization defenses execute the same sequence of instructions for each different diversification of the program, the only difference being the location of those instructions. Nonetheless, the location of instructions in memory can affect system performance, due to, *e.g.,* cache interactions. Indeed, previous work has evaluated the potential effects of caching through static WCET analysis of instruction caches [19], albeit for a simplified microarchitecture more representative of an embedded microcontroller instead of a modern multiprocessor with features such as virtual memory, multi-level caches, and out-of-order execution. That work reports WCET bounds as high as 1.8× that of undiversified binaries.

In this work, we present DART, a real-time address-randomization defense preserving predictable timing while greatly increasing the uncertainty of code locations in the virtual address space. DART randomizes code placement of basic blocks into virtual memory at page granularity specific to a set of placement constraints, based on a basic block's original location within a page, which fix the binary's cache color and intra-page alignment across the whole space of layout possibilities. This separation of the virtual address-space code placement with the physical mapping to cache behavior enables the randomization of potentially attacker-visible addresses while keeping cache color and alignment statically known, so that cache effects can be bounded by analysis of the binary alone. One limitation is that DART's predictability-preserving constraints restrict how basic blocks may be placed across pages (*e.g.,* blocks with overlapping intervals cannot share a page). We therefore present evaluations of the *entropy*, or number of unique randomizations possible. These results demonstrate that the entropy afforded by DART often exceeds other randomization schemes, or even the entropy of strong encryption schemes.

Thus, beyond its mechanism as a defense, we present a concrete security metric, the Shannon entropy of DART's layout search space, for analyzing the randomization induced by DART through a formal model of the constrained basic-block placements into virtual memory as the vertex coloring of an interval graph induced by the page-offset constraints set for consistent cache behavior. This way, we apply results from chordal graph theory to obtain a closed form count of the possible valid layouts for a given binary. Due to DART's random sampling from this space of possible layouts, this entropy serves as a formal metric for comparison of an adversary's uncertainty about code locations in virtual memory, allowing for comparison with conventional defenses.

This paper therefore makes the following contributions:

- We present DART, a real-time address randomization defense that randomizes code while fixing the binary's cache colors and intra-page alignment through a modified compiler, linker, and kernel chain.
- We present an analysis and empirical evaluation of DART's timing behavior on the TACLeBench and CoreMark-Pro suites, showing that its constraints bound layout-induced variance across the space of layouts.
- We formalize the layout space induced by DART by reducing the problem of basic block code placements to that of a problem of counting graph-colorings.
- We evaluate DART's security across the same TACLeBench suite binaries by reporting the entropy induced by the virtual memory layout and contextualizing it relative to common exploits and known defense baselines.

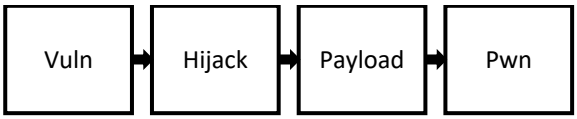


Figure 1 The four steps in an exploit. An underlying vulnerability, "vuln", is used to hijack control flow by controlling a code pointer, which redirects control flow to an attacker-chosen payload, that in turn "pwns" the system, *i.e.,* allows the attacker to win.

2 Background

Here we review background on security defenses and the attacker capabilities considered in this paper, as well as randomization, and caches. We begin with the threat model and the control-flow hijacking chain it enables, the existing randomization measures used to disrupt that chain, and common bypass techniques that motivate DART's page-level design.

Threat Model. We focus on adversaries who exploit memory-corruption and memory-safety style vulnerabilities to gain control over code pointers through control-flow hijacking primitives. Two sets of defenses form as a response to the capabilities of these adversaries. The first, and strongest, set of defenses attempts to eliminate the underlying bug through memory or type safety [39], but has historically seen limited deployment in production systems due to compatibility and performance costs, and has been used in testing environments [44, 47]. The second are designed to break one or both of the underlying primitives necessary for control-flow hijacking, including (i) the ability to corrupt a code pointer and (ii) sufficient knowledge of the address layout to construct an exploit payload [49, 1] and include the class of randomization defenses such as ASLR [49]. In this work, the threat model assumes the presence of a memory-corruption bug that allows an attacker to arbitrarily write memory, sometimes called a "write what where" vulnerability, and ensures that the adversary cannot have deterministic knowledge of the address-space layout.

2.1 Overview of Control-Flow Hijacking

Control-flow hijacking attacks seek to subvert a program for malicious purposes, including data exfiltration and giving an attacker control of the system. How exploit payloads are structured to achieve attacker goals is an entire field of research in its own right [6]. At a high level, attacks proceed along a four-step kill chain, illustrated in Figure 1, resulting in the system being pwn'd. The first step is exploiting an underlying vulnerability in the system, most commonly a memory-safety error such as a buffer-overflow or use-after-free vulnerability. This underlying vulnerability allows an attacker to control a code pointer, and thus what code is executed by the processor. Combined with knowledge of the code and data layout, the attacker is able to construct and inject their attack payload, which is then executed. Once the payload has been executed, an attacker has pwn'd the system.

There are two key primitives in the attack kill chain for control-flow hijacking attacks: (i) attacker control of a code pointer, and (ii) knowledge of the code and data layout to construct the exploit payload. These primitives are commonly targeted by defenses to disrupt the kill chain, causing attacks to evolve to bypass newer defenses and operate with greater constraints. In particular, the deployment of Data Execution Prevention (DEP) [37] and Write XOR Execute ($W \bigoplus X$) [40] has virtually eliminated code-injection attacks, and caused the rise of code-reuse attacks.

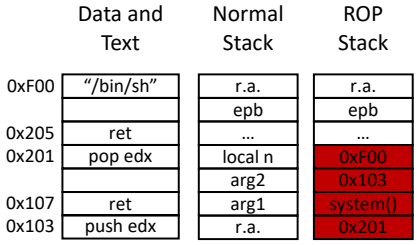


Figure 2 Code reuse attack example. Return address is overwritten to divert control to attacker chosen "gadgets", chained together by attacker chosen return addresses on the stack. Ultimately `system("/bin/sh")` is executed.

Code-Reuse Attacks. The current state-of-the-art software attacks all rely on *code-reuse* [43, 8, 45, 26] to bypass DEP. There are many code-reuse variants, all of which rely on "gadgets", *i.e.,* code snippets that perform a specific computation or operation, and which can be chained together to give an attacker control over a process. Figure 2 illustrates how such gadgets work in practice. The left column shows excerpts from the data and text sections of the program. In particular, the string `"/bin/sh"` is at address `0xF00`,[1] and executable bytes at addresses `0x107` and `0x205` are the `ret` instruction.[2] The `ret` instructions are preceded in this example by `push` and `pop` instructions that move a value from the stack to register `edx` and back. The middle column shows the stack under normal execution for the 32-bit *System V* calling convention. Assume an attacker can overwrite values on the stack due to, *e.g.,* a memory-corruption vulnerability. The attack payload to perform a code reuse attack and open a shell is shown in the right-hand-side column. The first return address is overwritten to `0x201`, causing the next value on the stack, the address of `system()`, to be popped into `edx`, before sending control flow to `0x103`. The gadget at `0x103` pushes `edx` to the stack, and then returns, effectively calling `system()` with the argument `"/bin/sh"` as `0xF00` is the next value on the stack, and thus the first argument.[3] This style of attack is known as return-oriented programming (ROP) as returns are used to chain together gadgets.

Once an attacker has identified gadgets, they must divert control flow to their chosen gadgets. Control-flow hijacking attacks are executed by corrupting code pointers, *i.e.,* pieces of data that point to executable memory. Code-reuse attacks are classified by the type of code pointer corrupted to hijack control flow, such as return addresses in Return-Oriented Programming (ROP) [45], addresses for indirect jumps in Jump-Oriented Programming (JOP) [11], indirect calls in Call-Oriented Programming (COP) [8], or C++ virtual tables in Counterfeit Object-Oriented Programming (COOP) [43].

Randomization Defenses. Randomization defenses seek to mitigate code-reuse attacks by changing the virtual addresses of instructions, thereby preventing attackers from constructing gadget chains. Randomization defenses are defined by the level at which they randomize: library level, *e.g.,* ASLR [49], function level, *e.g.,* selfrando [14], or basic block (BBL) level, *e.g.,* Compiler-assisted Code Randomization (CCR) [29]. Instruction-level randomization [27, 25]

[1] Note that in practice, `libc` does actually contain this string.
[2] Note that x86 is a variable-length instruction set, so processors do not impose alignment constraints on instructions. Consequently, these may not be `ret` instructions emitted by the assembler, but bytes from the end / beginning of other instructions that happen to encode a `ret` [26].
[3] The `pop` / `push` gadgets are effectively a no-op in this example, but serve to illustrate how gadgets can be chained together to form attacks.

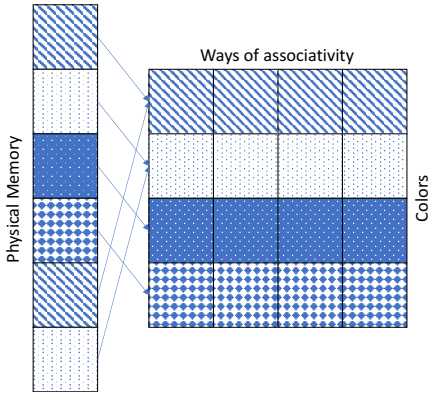


Figure 3 Cache architecture and coloring.

moves instructions by inserting No-operation (`NOP`) instructions, in contrast to the other techniques that can re-arrange code in arbitrary configurations, aided by the control-flow instructions at the end of functions or BBLs. Consequently, we view CCR as the state of the art for code randomization, and adapt it for DART.

Partial Pointer Overwrites. A common assumption is that attackers must overwrite or learn an entire code pointer (e.g. a return address, function pointer, or a Global Offset Table (GOT) entry) for control flow hijacking. To counteract these randomization defenses, such as ASLR, a class of software vulnerabilities allowing an attacker to redirect control flow without knowledge of the layout of virtual memory has emerged. In practice, many memory-corruption vulnerabilities allow for writing with byte-level granularity, which lets an attacker modify the least significant byte(s) of a pointer while leaving the higher order bytes unchanged. Typically code pages are aligned, one after another, and thus these low bytes correspond to an offset within the code or libraries, which can be controlled to redirect execution to a gadget or executable segment without knowledge of the randomized high-order bytes. Often called a *partial pointer overwrite*, this technique has been used to bypass randomization defenses [54, 24] and, in later sections, is used to motivate DART's page-level randomization in which knowing the right page becomes part of the attacker's guessing problem, and is shown to be computationally infeasible.

2.2 Caching and Virtual Memory

Randomization affects the layout of instructions in virtual memory, which can (though need not) affect the layout of instructions in physical memory as well, depending upon the randomization scheme. Physical addresses are used to index into the cache subsystem, and therefore, altering the layout of physical addresses can in turn change the cache behavior at runtime. To fully understand these implications, we first review cache properties that are relevant to memory-randomization performance impacts.

Modern processors have multiple cache levels. The first cache level (L1) is dedicated per processor, and is *split* between instructions (L1-I) and data (L1-D). Because we are evaluating code randomization, we focus on the instruction cache. At subsequent cache levels, code and data are *unified* and included in the same cache.

While the cache operates at the granularity of cache lines, for example 64B, the OS controls memory allocations at the granularity of a page, which we assume to be 4096B (64 cache sets). Physical pages of memory are assigned *colors* in such a way that pages that

map to the same cache sets, and thus conflict, are assigned the same color. This is shown in a simplified exemplar cache architecture in Fig. 3. The OS therefore has some degree of control over the cache conflicts through its choice of physical-memory allocations.

When two or more pages are allocated to the same color, memory references from those two pages may conflict in the cache. However, the specific conflicts that occur at runtime, if any, depend upon the access patterns, as well as the layout within the individual pages. A randomization defense that is applied at a sub-page granularity, such as CCR [29], therefore affects both the virtual and physical memory layout, which in turn changes the potential cache conflicts at runtime. Crucially, on stock Linux this color assignment is a property of the running system at allocation time, not of the binary itself, so two executions of the same binary may map to different cache configurations.

3 DART Requirements

Randomization and moving-target defense (MTD) techniques improve security without inflating binaries, requiring additional hardware, or consuming significant memory, doing so with modest average-case computational overhead (typically 0–5% [49, 52, 33]). For RTES, however, the primary constraint is not average-case performance but predictable worst-case performance. We define *predictability* in the sense standard to real-time systems. Specifically, a system exhibits predictable timing when its execution-time variability across runs is bounded such that the worst-case execution time (WCET) can be analyzed with adequate tightness. For example, in cache analysis, if it is not known which cache accesses conflict, a sound analysis must conservatively assume that they all do, leading to highly pessimistic worst-case execution-time (WCET) estimates. In this paper, we present an address-randomization defense, DART, that achieves predictable timing in this sense, establishing execution patterns that are amenable to timing analysis.

Because DART does not modify the dynamic instruction trace, the same instructions are executed in the same sequence, and only their memory locations change. Consequently, the primary mechanism by which different layouts introduce timing variability across diversified binaries is the instruction cache, as the spatial placement of instructions dictates which cache sets are accessed and which lines may conflict. To address this, DART fundamentally decouples the address uncertainty required for virtual-memory randomization from timing-relevant cache behavior. Specifically, it encodes the binary's cache color and intra-page alignment statically during link time, rather than permitting the operating system (OS) allocator to determine these properties dynamically at load time.

To make these constraints explicit and well-defined, we isolate the effects of code movement into three distinct dimensions that impact security and timing:

- **Virtual Address (VA) page remapping:** Remapping virtual pages without altering their corresponding physical memory locations or cache assignments;
- **Physical Address (PA) color:** Mapping virtual pages to different physical pages, which alters the cache sets they occupy and introduces potential conflicts; and
- **Intra-page offset:** Adjusting the offset of code structures (*e.g.,* basic blocks or functions) within a page, which modifies their cache-line alignment and conflict footprints.

Timing variability in real-time systems arises from numerous sources, including data caches, branch predictors, DRAM refresh, interrupt arrivals, and other microarchitectural behaviors. Bounding or eliminating any of these sources improves predictability. Code-layout randomization introduces a dominant new source: layout-to-layout variability in instruction-cache behavior. This variation arises because, although diversified binaries execute the same

instructions in the same sequence, they place these instructions on different physical pages and at different intra-page alignments, altering which cache lines conflict. Consequently, we define *predictable instruction-cache behavior* as a property where the layout-induced component of instruction-cache behavior, and therefore of execution time, exhibits bounded variability across the entire space of layouts generated by the defense. Other sources of timing variability are unaffected by code-layout randomization and are outside the scope of this paper. DART achieves this property by constraining **PA color** and **Intra-page offset** such that cache color and intra-page alignment are determined solely by the binary. This constraint fixes the conflict set of every code page at link time, rather than leaving memory mapping to the OS allocator at load time. In contrast, probabilistic timing analysis achieves timing guarantees by characterizing this variability statistically over a sampled population (Sec. 7). We discuss compatibility with concrete timing-analysis tools at the end of this section.

VA page remapping. Some defenses, such as address-space layout randomization (ASLR), move whole segments of code by changing page tables to map that code into different sections of the virtual-address (VA) space. This remapping does not alter the physical layout of the pages in memory. How these pages map to physical memory is an independent physical-address effect discussed under PA color. Because the VA space is large (2^{48} bytes on x86_64), shifting pages within the virtual-address space offers significant opportunities for achieving randomization entropy. DART employs a page-level randomization model in which code pages are shuffled among multiple virtual pages of the same color. This structure mitigates code-reuse attacks by reducing the feasibility of partial pointer overwrites or arbitrary writes.

PA color. How a page maps to physical memory affects which cache sets it can occupy (recall Fig. 3). Standard operating systems like Linux, however, lack interfaces to control physical memory allocation or cache set placement. Consequently, the exact mapping to the cache is determined dynamically by the kernel allocator (e.g., the buddy allocator), introducing significant run-to-run and layout-to-layout timing variation. As a result, timing becomes unpredictable, forcing sound timing-analysis tools to conservatively assume that all pages conflict on the same cache set.

Note that the Linux kernel caches executable files and metadata (inodes) in the page cache. Consequently, successive executions of a process may reuse these cached pages. While this page-cache reuse might suggest run-to-run determinism, it is unreliable; the executable can be evicted at any time to reclaim memory. Furthermore, when the program is initially loaded, the physical allocations remain uncontrolled and non-deterministic.

To address this unpredictability, DART enables explicit, ELF-encoded cache coloring of code pages, replacing the OS allocator's dynamic page assignment with a static coloring configuration fixed at link time. DART encodes the target physical cache color directly into the ELF header of each page. The operating system loader then allocates and loads each virtual page into a physical page matching that designated color, preserving cache-line alignment.

Intra-page offset. Fine-grained randomization defenses operate at a sub-page granularity, altering the layout of instructions and changing the alignment of functions and basic blocks (BBLs) within their pages. For example, if BBL A is located at the beginning of a page and BBL B is located at the end, they map to different cache lines and do not conflict, even if their pages share the same physical color. However, if B is shifted to the beginning of its page,

it will map to the same cache line as A, introducing conflicts. Fine-grained randomization techniques, such as CCR [29], permute code structures within the `.text` section. These permutations alter the relative offsets of instructions, introducing uncontrolled cache conflicts even when page coloring is preserved.

This shift in code alignment manifests in two distinct effects: (i) it alters the set of co-located code structures that map to the same cache sets, and (ii) it can change the cache-line alignment of individual blocks, potentially increasing the number of cache lines they span. For example, consider a BBL whose size is less than a cache line. If the block is aligned to the start of a cache line, it fits entirely within a single line. If it is shifted such that its instructions cross a cache-line boundary, it occupies two cache lines. This shift increases the block's cache footprint and can necessitate an additional cache line fetch at runtime. Consequently, fine-grained randomization must account for both intra-page offsets and cache-line alignment to preserve predictable timing. To eliminate this source of timing variability, DART preserves the static intra-page offsets and cache-line alignment of basic blocks. This design choice maintains cache-line determinism and allows static analysis to precisely bound cache conflicts, while still realizing significant security entropy through page-level randomization.

In Sec. 6, we quantify the security entropy achieved by DART's page-level randomization (VA page remapping). In Sec. 5, we empirically evaluate how constraining physical page coloring (PA color) and preserving intra-page offsets (Intra-page offset) yields predictable timing behavior. Specifically, layout-induced timing variance is bounded by a monochrome coloring-conflict configuration and evaluated using the coefficient of variation (CV) across randomized layouts.

Compatibility with Timing-Analysis Techniques. By establishing instruction-cache color and intra-page alignment as static properties of the binary, DART renders cache behavior more amenable to existing timing-analysis methodologies. Static worst-case execution-time (worst-case execution time (WCET)) analyzers, such as aiT [20] and OTAWA [4], construct Must and May cache state abstractions using abstract interpretation. Because DART fixes the conflict set of every code page at link time, these abstractions can be tightened significantly, avoiding the pessimistic assumption that all pages conflict. Similarly, for measurement-based timing analysis, DART tightens empirical execution-time bounds by eliminating OS-allocator-induced run-to-run color drift. Finally, DART is compatible with measurement-based probabilistic timing analysis (MBPTA) [9, 31, 10], which requires independent and identically distributed (i.i.d.) timing samples. DART's combinatorial layout space generates exactly such a randomized population, while fixing the specific layout of the deployed binary to eliminate dependence on runtime operating system state. A full integration with any of these toolchains is beyond the scope of this work; instead, we evaluate the impact of DART on instruction-cache behavior empirically in Sec. 5.

4 DART Implementation

We present DART, a real-time address-randomization defense designed to increase uncertainty about code locations in virtual memory while preserving predictable timing by binding cache color and page alignment as static properties of the binary. Doing so requires enforcing per-page cache-color assignments at load time and controlling the physical pages that code gets allocated to, necessitating changes to the loader, to assign code to appropriate pages, and the kernel allocator, so the loader can obtain the desired pages. DART requires

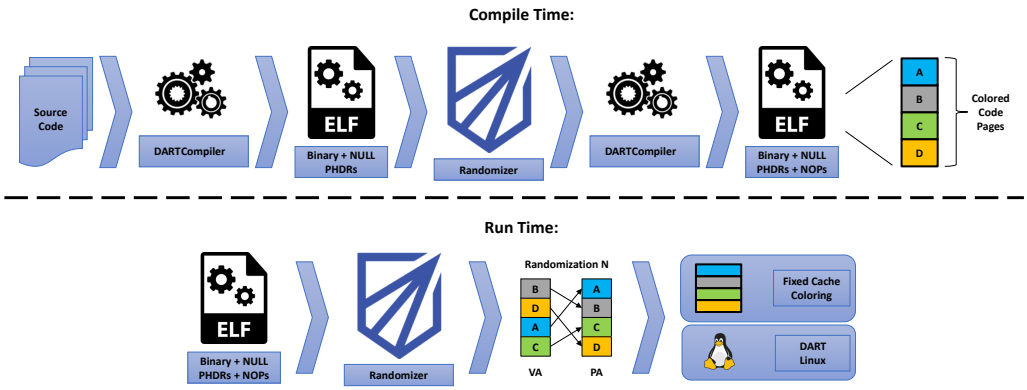


Figure 4 Overview of the DART defense framework.

communicating the desired per-page code-cache layout to the loader, which we implement through modifications to the executable and linking format (ELF) file format, implemented in the linker as it emits the relevant parts of the ELF file. DART also requires the ability to randomize code generating diversified layouts, which DART does at the BBL level, which we implement via compiler and binary-rewriting support.

Although DART's implementation includes modifications of multiple layers of the system stack, changes at each layer are localized and additive rather than interfering with the existing implementations and functionality. Kernel changes, for example, are confined to ELF program-header parsing in the loader and optional extensions of the buddy allocator API. The compiler changes are confined to a late-stage LLVM pass allowing for basic-block splitting and expansion of the .text segment as well as a Gold linker hook for per-page program headers. This configuration of constrained additive and opt-in changes makes DART deployable in real-time Linux environments particularly where the underlying stack and kernel are already under active real-time maintenance.

An overview of DART's design that meets these requirements is presented in Figure 4. DART's compiler toolchain produces an ELF file that has been randomized and has cache coloring information embedded. Then, at run time, DART's Linux extension provides the requested coloring. DART builds on top of an existing BBL-level randomization tool, CCR [29]. DART extends CCR to support page-aware diversification with fixed offsets. DART builds on top of CCR, which uses `llvm-9.0`. CCR depends on `binutils-2.37` in implementation. The DART loader and allocator extensions modify `Linux 5.4.0`.

4.1 Compile Time

The compile-time portion of DART takes in application source code, and outputs a binary that is not only randomized, but also has control over cache alignment and code color at the BBL level. DART exposes configuration flags controlling how the binary is randomized, the cache alignment, and the cache coloring used for deployment and ablation in evaluations. Based on the settings of the flags, additional code pages may be required to allow for randomization while still maintaining the desired alignment and code-color properties.

The DART compile-time tool chain consists of three elements. The *randomizer*, extended from CCR, calculates and applies a randomized code layout that respects the defense constraints on whether to randomize, cache alignment, and cache coloring. We further

extend *LLVM* and the *Gold* linker to enable the construction of ELF binaries consistent with DART's runtime enforcement requirements. To do so, we re-purpose an unused field in the ELF program header (PHDR) to encode the desired color for a given code page.

To support constrained diversification of a binary to shuffle BBLs while preserving offsets, DART includes a compile-time flag to determine the binary's expansion, which is filled with invalid instructions or dead code (code that is never executed). The linker assigns an ELF PHDR per code page. Doing so enables specifying a cache color for every code page via the co-opted field in the PHDR. A randomizer script can then be run to rearrange the BBLs within the `.text` section and create different randomized binaries without recompilation from source.

4.1.1 Randomizer Script

The randomizer script is written in Python and used by CCR to perform function- or BBL-level randomization. DART extends the basic-block-level randomization to implement its diversification policy while maintaining the aforementioned fixed-offset and coloring constraints. The mechanics of moving BBLs, maintaining metadata, fixing up code pointers and function addresses, *etc.* after randomization are unchanged from CCR [29]. The randomizer script is responsible for managing code cache colors as well as alignment within DART. The randomizer script realizes the code cache colors through modifications to the ELF PHDRs.

Destraddling. DART requires that every BBL reside entirely on a single page. Doing so allows the virtual addresses of code pages to be independent; without this requirement if a BBL resided on two code pages they would have to be contiguous in virtual memory. This preserved independence of page selection ensures the correctness of the placement model used for determining the security induced by layout availability. To implement this, the randomizer script performs a pre-transformation of the code base before randomization, which we refer to as "destraddling." BBLs that "straddle" two pages are moved to the end of the code section, and placed sequentially on a new code page (or pages).

Basic-Block Splitting. Programs occasionally contain BBLs larger than a single page. This presents a problem for destraddling, as any BBL larger than a page will straddle at least two pages regardless of placement. To prevent this, an LLVM pass was developed to split large BBLs. The pass runs after a program is lowered from LLVM IR to machine code, but before it is emitted as an object file. Basic blocks and their fallthroughs are identified and their sizes in bytes are estimated based on the number of instructions they contain. If a basic block is estimated to be larger than the maximum BBL size set with the `--max-bb-size` flag, it is split in half by inserting a non-conditional jump instruction from the first half to the second. The results of this are then recursively checked for size and split accordingly. Though this method of splitting inserts jump instructions that would not have been present in an unmodified binary, in practice the vast majority of basic blocks are unmodified, and the performance effects of splitting at the size of a 4096-byte page are negligible [35].

DART additionally supports enabling randomization such that every BBL is placed at the same offset from the start of the page between randomizations, which stabilizes cache-line alignment effects allowing for the ultimate goal of statically characterizable timing with high address uncertainty. To enable this feature while achieving a reasonable number of randomizations, DART accepts as an argument a user-specified number of code pages to add per color.

At diversification time, DART constructs a valid layout instance with the following randomization algorithm: the randomizer attempts to place each BBL in turn on a randomly selected page of the correct color. If there is a conflict with another BBL already on that page, then the randomizer randomly selects another page and so on recursively until either the BBL is placed or all pages of the correct color are exhausted. If no eligible page can be found, then the randomization algorithm resets all assignments and tries again. The number of attempts to create a valid randomization of the BBLs is configurable as an argument to the `shuffle_BBLs` function in `dartRandomize.py`. Increasing the size of the code section with the aforementioned compile-time expansion increases the likelihood that an offset-preserving randomization can be quickly found.

4.1.2 LLVM Extension

The LLVM extension serves to add the necessary code pages for the aforementioned offset-preserving randomization. The extension is controlled by a compiler flag that specifies how many bytes need to be added to the `.text` section of the ELF. The number of bytes specified is rounded up to the nearest multiple of 16. The inflation flag controls an LLVM pass that runs while lowering from LLVM IR to assembly.

4.1.3 Gold Linker Extension

The linker is responsible for determining how the loader should set up a program in memory. This information is encoded into ELF PHDRs. Each ELF PHDR details metadata about where in the ELF file the data it controls resides, what virtual address to load it at, and what permissions to assign the memory, *e.g.,* read write, read execute, or read only. Constituent elements of the program, *e.g.,* `.text`, `.data`, *etc.* are grouped by access permissions, and then loaded by one program header.

By default, one PHDR is emitted for the `.text` section as it is a contiguous chunk of read/execute memory. DART modifies the linker to emit one PHDR per code page in the `.text` section. The number of PHDRs required is determined by the randomizer and passed to the linker using the `--null-phdrs` flag – the additional PHDRs are initially null until set by the second pass of the randomizer script.

To encode the color of a page within its PHDR, we leverage an unused field. PHDRs contain a field for specifying the physical address – `p_addr` – to load the PHDR's data. This field is ignored by default in the Linux Kernel, and is usually set to the same value as the `vaddr` value by the linker. We leverage this unused physical address identifier to provide coloring information to the loader.

4.2 Run Time

DART includes kernel extensions to enforce code cache-coloring for code pages. Specifically, we extend the Buddy Allocator API to allow the caller to specify the desired color of the underlying physical page that is returned. DART then modifies the Loader to use this API when loading applications in memory.

4.2.1 Allocator

DART extends the Buddy Allocator API by adding the `alloc_page_colored` function, which has an additional argument `color` compared to the standard `alloc_page` function. The internal `colored_page_list` can be pre-populated with colored pages to ensure all binaries

can be loaded without fetching additional pages and so page-color allocations do not fall back to opportunistic search at load time. To fulfill the requested page color, the allocator first checks this internal `colored_page_list` to see if it already controls a page of the desired color. If it does, that page is returned. Otherwise, the allocator acquires a new page in the usual way. If the new page is of the desired color, it is returned. If not, it is added to the list of pages controlled by the allocator. The allocator acquires pages in this manner until a page of the specified color is found and returned. This is further discussed in Section 5.

4.2.2 Loader

The DART modified loader parses the binary's PHDRs for color information. Doing so requires extending the number of PHDRs supported by the loader. PHDRs are 56 bytes, and the number of PHDRs in an ELF is an eight-byte field. The mainstream Linux kernel interprets the number of PHDRs as the number of PHDR *bytes*, resulting in $2^8/56$ PHDRs being supported. DART changes this interpretation to be the number of PHDRs, matching the ELF standard [13], and so can support 2^8 PHDRs, ample for our purpose. To maintain DART's coloring invariants and prevent unexpected results due to caching, we also modified the behavior of the loader should a page already be loaded in the cache. If this is the case, the loader will check if the color of the page is correct – if it is, the loader continues. If it is incorrect, it will reallocate a new page of the correct color, then migrate the existing page to the new page, freeing the cached page afterwards if necessary.

5 Evaluation

DART's contribution is a defense design with timing certainty and security metrics suited to RTES. We therefore evaluate DART along both critical axes. First, we analyze execution-time behavior across diversifications and show that the constrained changes make layout-induced timing variance attributable to the binary rather than to runtime OS state. We then quantify security by modeling valid layouts under DART's constraints from a combinatorial perspective and compute the resulting layout entropy. This yields a measure of the search space faced by an attacker and allows for strict comparison with existing randomization baselines such as traditional ASLR (Sec. 6).

In this section we use DART to explore how the code movement that results from randomization, as discussed in Section 3, affects code caches and thus real-time performance and predictability. In particular, we seek to answer these fundamental research questions:

RQ1: How sensitive is execution time to the layout-induced intra-page alignment changes, and what WCET overheads arise when page-alignment is unconstrained across the diversifications?

RQ2: To what extent do DART's constraints in preserving basic-block offsets and enforcing cache coloring bound the layout-induced variability of instruction-cache behavior across the space of layouts produced, and do they admit a measurable bound on the coloring-conflict component?

RQ1 seeks to quantify the impact of changing the binary layout in the cache at page color and sub-page granularities. RQ2 explores how these effects can be mitigated by keeping page coloring and page offsets constant between randomizations. We begin by leveraging DART to compile and evaluate the TACLe and CoreMark-Pro benchmark suites with varying degrees of control over these variables.

5.1 Experimental Setup

In our evaluation, tests are performed on a Beelink Mini S12 mini PC equipped with a 4-core 64-bit Intel N100 x86_64 processor. Each core has an independent 64 KiB, 8-way L1-I cache as well as a 32 KiB, 8-way set-associative L1-D cache. The L2 cache is a 2 MB, 16-way non-inclusive associative cache, which is unified between data and instructions and shared across cores. The last-level cache (LLC) is 6 MB, 12-way associative, and shared between all cores as well as the integrated graphics chip.

CCR, which we build upon, only supports x86, so this platform was selected as the most impoverished modern x86 consumer platform we could find. This device is running Ubuntu 20.04.4 Long-Term Support (LTS) with our DART extension to the 5.4.0 Linux Kernel. With 4096 bytes per page, we need $6\,\text{MB}/12/4096 = 128$ colors to color the LLC; the LLC cache set is determined by the 7 lowest-order bits of a physical page number. Since the L2 and L1-I caches need 32 and 2 colors respectively, coloring at the LLC ensures color control at all levels of the cache.

DART implements two coloring schemes. Under *monochrome*, all BBLs are assigned a single color, providing a deliberate adversarial configuration that bounds the coloring-conflict component of cache contention. Under *striped*, available colors are assigned sequentially across pages, spreading code across all cache sets; this is not an optimal assignment (which would require workload-dependent cache-conflict analysis), but suffices to validate that DART's coloring enables predictable, performant timing.

Additionally, when a binary is shuffled in vanilla CCR, the set of conflicting BBLs can change and if frequently used BBLs sit at the same offset within a page and map to the same color, they may frequently evict one another. We include a build-system flag, used with BBL splitting, that limits the region of each page in which basic blocks may be placed (e.g., the first 512 bytes), giving an evaluation knob for exploring worst-case cache contention in highly constrained layouts.

To capture benchmark timing data, we augment benchmarks by adding calls to the Linux `clock()` function at the beginning of `main()` and just before returning. In order to minimize system noise and interference, we set each benchmark to the highest system priority, and pin all benchmarks to a dedicated CPU core. Frequency scaling is disabled for all experiments. To isolate cache effects to those from benchmark processes, the Intel Cache allocation technology (CAT) software is used. CAT enables one to specify which ways of L2 cache are accessible to which cores. In our setup, two ways of the L2 cache are isolated and dedicated to the benchmarking core. To isolate performance results from the effects of demand paging, the Linux `mlockall()` function is used to ensure that all code pages are loaded into memory before timing begins. We note this is commonly done in RT applications to avoid page faults during real-time computations. The CoreMark-Pro testing harness is adapted for compatibility with our testing methodology.

We perform two sets of experiments. In the first (RQ1), we quantify the worst-case cache effects of changing cache alignment under randomization, approximating the range of execution times that can arise as layout and cache contention vary. In the second (RQ2), we isolate the effects of page offsets and page coloring on cache behavior and execution time, and showcase how DART controls them across runs.

5.2 RQ1: Cache Alignment

The first set of experiments explores how alignment affects performance. Randomization without control for page offsets or cache coloring can produce binaries whose basic blocks' cache alignment and conflict sets vary significantly between randomizations. A hypothetical worst-case randomization, then, would have all of a program's most often used BBLs placed

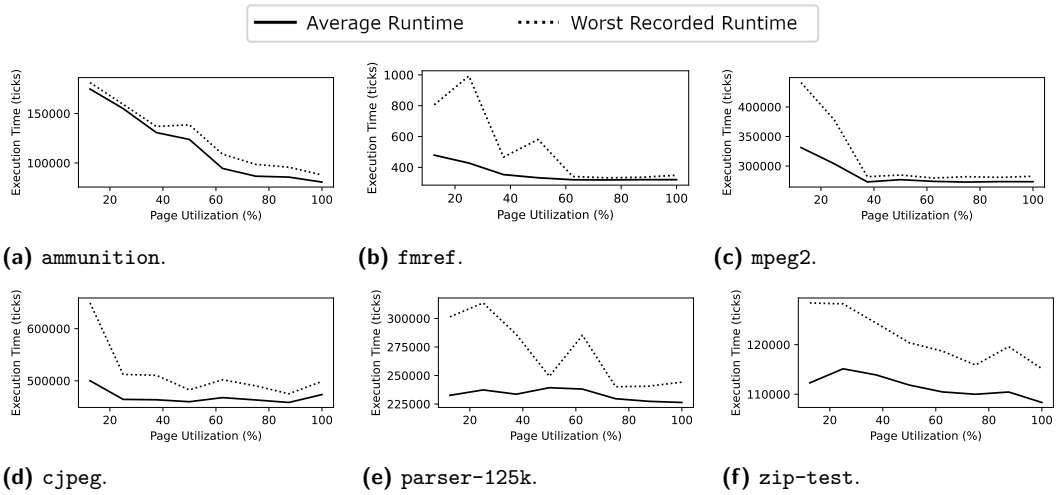


Figure 5 Runtimes for Benchmarks With Varying Levels of Page Utilization.

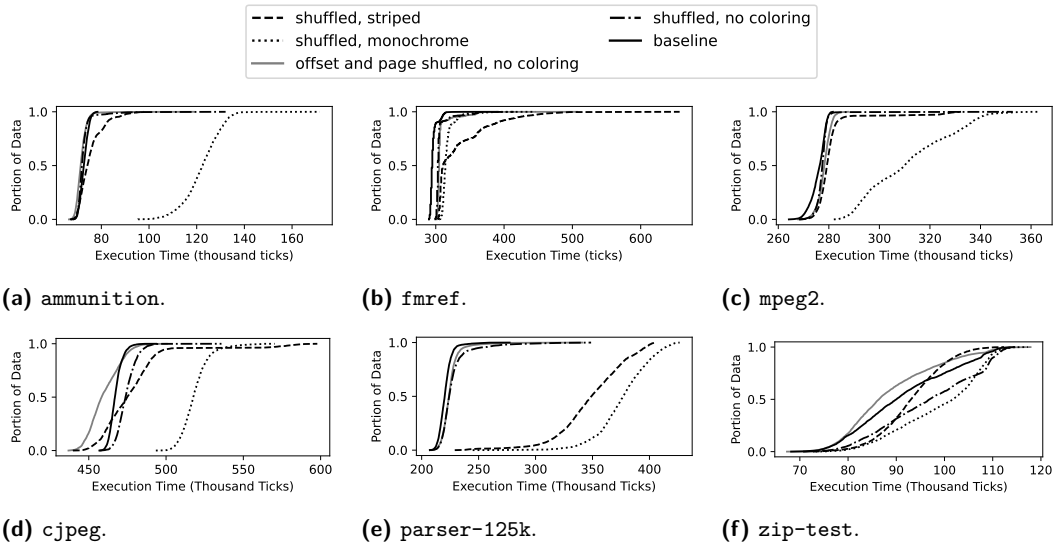
at similar page offsets, and on pages of the same color, maximizing cache contention. Without a system such as DART to constrain these conflicts across runs, future runs of a binary may induce worse layouts and worse cache conflicts despite any real-time guarantees that may be desired. To quantify this effect, we force this worst-case scenario using DART. To do this, we leverage the ability in DART to generate binaries with varying amounts of usable page space. We generate eight binaries with between 12.5% and 100% page utilization. Each page of each binary is assigned to the same color, to maximize cache collisions. We run each 100 times for each benchmark for a total of $8 \times 100 = 800$ data points per benchmark binary. To control for the potential performance effects of BBL splitting, each binary is compiled with a maximum BBL size of 512 bytes.

Figure 5 shows the results of this experiment on six representative benchmarks.[4] The x-axis shows the percentage of page utilization for each binary, and the y-axis shows average and worst-case runtimes. The dominant trend is that binaries allowed to use the whole page performed better than those given less space, in some cases by margins exceeding 100%. In most panels of Figure 5, worst-case performance closely follows average performance, suggesting that such strict control over coloring and page layout largely restricts the effects that cache can have on performance. The `parser-125k` benchmark is an exception, exhibiting a non-monotone average runtime across utilization levels. Across the suite, the trend holds, and worse virtual memory layouts with greater cache contention performed worse than layouts that were allowed to be more sparse, and so randomization defenses with no regard for cache behavior can unpredictably induce worst-case layouts with maximum cache contention.

5.3 RQ2: Coloring and Randomization Effects

Having established that layout drives timing variance, we now ask how DART's offset and color constraints reshape that variance. We utilize DART to generate four types of randomized binaries, and compare these to one "baseline" binary. Three variants preserve

[4] Benchmarks were chosen to give a representative view of the trends that appeared in the TACLe benchmarks (top row), as well as examples from CoreMark-Pro (bottom row). Due to space constraints, we omit the remaining benchmarks, but the full datasets are available in the online appendix at `https://my.vanderbilt.edu/bryancward/publications/`.



(a) `ammunition`. **(b)** `fmref`. **(c)** `mpeg2`.

(d) `cjpeg`. **(e)** `parser-125k`. **(f)** `zip-test`.

Figure 6 Runtimes for Benchmarks Across Coloring and Randomization Schemes.

Table 1 Per-scheme distribution-shape statistics relative to the un-randomized baseline, summarized as medians across all 32 binaries. *intra CV* is run-to-run jitter for a fixed BBL layout; *inter CV* isolates BBL-placement variance across randomizations; *worst over.* is the max cycle count vs. baseline median; *tail>10%* is the fraction of trials beyond baseline median + 10%. Per-benchmark numbers behind the six panels are reported in the online appendix at `https://my.vanderbilt.edu/bryancward/publications/`.

Scheme	**intra CV** (%)	**inter CV** (%)	**worst over.** (%)	**tail** $>10\%$
shuffled	4.7	0.76	45.6	5.0%
offset+shuffled	5.4	0.64	37.7	5.2%
monochrome	8.5	0.92	101.7	78.6%
striped	6.7	0.96	78.2	35.2%

BBL page offsets between randomizations and differ only in coloring: (i) *monochrome*, where every page is assigned the same color, (ii) *striped*, where colors are assigned sequentially across pages, and (iii) *no coloring*, with no modified coloring. These approximate an adversarial coloring-conflict, dispersed, and control case cache layout, respectively. Page-offsets for BBLs are kept constant between randomizations, so randomization is achieved by placing BBLs on pages of the correct color. For a minimal-control comparison, we also compile and randomize a binary without maintaining page offsets and apply no coloring. Lastly, the "baseline" binary uses no randomization or coloring.[5] 25 randomizations are generated for each benchmark, and are run 100 times (2500 runs total per benchmark).[6]

Results are shown as cumulative density functions (CDFs) in Figure 6 while Table 1 provides some aggregate statistics. Notably, we find that inter-randomization Coefficient of Variation (CV)[7] under striped is approximately that under shuffled (median 0.96% vs

[5] For consistency, any BBL splitting done in the randomized binaries is also done in the baseline binary.
[6] The baseline scheme is not randomized, so it is simply run 2500 times. A subset of the CoreMark-Pro benchmarks (radix2, linear-all, loops-all) take exceptionally long to run, 2500 runs of which would have taken an estimated six weeks. For these benchmarks, we were limited to 10 runs per randomization, instead of 100.
[7] Coefficient of Variation is a unitless measure of relative dispersion defined as standard deviation divided by the mean.

0.76% respectively) and so DART is effectively encoding color into the binary at no aggregate variance cost. Additionally, the monochrome coloring scheme had median 78.6% of trials at more than 10% of the baseline, giving a measurable bound on the coloring-conflict component of WCET; intra-page placement variance under striped layouts is a separate component, and for a small number of binaries (*e.g.,* `parser-125k`, `fmref`) individual striped layouts can exceed the monochrome median in spread.

For every randomized scheme, the intra-randomization CV is several times larger than the inter-randomization CV and so two runs of the same randomized binary differ more than two different randomizations of the source differ on average. Thus, a real-time analysis of a DART binary is not analyzing a moving target and since the cache-color of the binary is set at link-time, any remaining timing variation is the same jitter that the binary typically exhibits rather than any layout-induced variability. This is also visible in the CDFs of Figure 6 where the curve for each scheme is tight and vertical separation between schemes is larger than spread within schemes. Additionally, we note that on benchmarks whose cache footprints fit within their available colors (*e.g.,* `cjpeg`), the striped distribution closely tracks the un-randomized baseline despite the security benefits and address-space entropy induced (Sec. 6). On benchmarks with high cross-color BBL conflict density (*e.g.,* `parser-125k`, `ammunition`), striped widens compared to the OS-allocator, and the inter-randomization CV stays under 1% so the widening is repeatable across runs of any given binary. The OS-allocator schemes can produce comparable widening on the same benchmarks, but only based on system state at allocation time which is not tractable with static analysis so DART transforms this from an unknown at deployment to per-binary properties seen at build time.

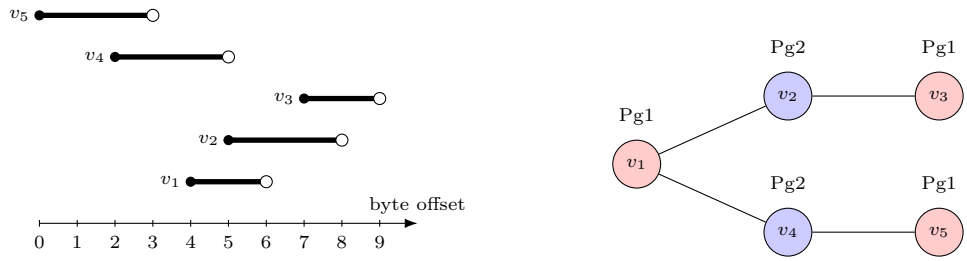
A non-DART binary's coloring and other properties depend on the state of the page cache, free lists, and concurrent allocations, while a DART binary executes with the same coloring and BBL page-offset on every run on every machine. This dependence on the OS allocator forces static WCET analysis to fall back on conservative abstractions, whereas DART shifts the cache color and intra-page alignment to properties of the binary that are fixed at link time and statically characterizable. Combined with the empirical coloring-conflict bound provided by monochrome, this tightens the inputs available to the timing-analysis approaches discussed in Sec. 3.

6 Security Evaluation

With cache effects on performance and WCET shown to be statically characterizable under DART, we now evaluate DART's security and capabilities in preventing control-flow hijacking. DART's model of page-level randomization splits a program into its corresponding basic blocks, assigns each basic block a page color, and places the basic blocks at fixed offsets within pages of their corresponding page color. For the purpose of mathematical formalism, a randomization instance chooses which of the m_c available pages of page color c a particular basic block of page color c will be placed on, with uniformly distributed probability. Two basic blocks with overlapping intervals cannot be placed on one page. For the purposes of evaluating security, it is important that we have a concrete metric for measuring the difficulty an attacker faces when attempting to control a particular word in memory from anywhere a vulnerability may occur, or the "entropy" of the randomization that DART induces.

6.1 Entropy Derivation

We will show a closed-form solution for the number of combinations of basic-block placements in the available pages by formulating the problem of placing basic blocks in pages as one of finding proper vertex colorings of an interval graph over the basic blocks, applying known



(a) Basic blocks as half-open intervals. **(b)** Interval graph and a valid 2-labeling.

Figure 7 Basic blocks induce an interval graph where overlapping blocks form edges, and an assignment of blocks to pages corresponds to a vertex-labeling.

results in chordal graph theory. To avoid confusion between vertex coloring and the standard page coloring, we will use the nonstandard terminology "vertex labeling" and "graph labeling" when approaching the coloring problem.

Model. Each basic block b_i has an associated page color c_i, a fixed-byte offset within a page o_i, and a size s_i. Thus, it will occupy the half-open interval $I_i = [o_i, o_i + s_i)$ within a particular page. Since pages and the basic blocks assignable to them depend on page color, we first consider the problem with only a single page color c where every basic block and every available page are of page color c. Then, let $V_c = \{b_i \mid c_i = c\}$ be the set of basic blocks and m_c the number of available pages of color c.

Example. We begin a running example to demonstrate the various components of this proof (Fig. 7). Consider the set of basic blocks $\{v_1, v_2, v_3, v_4, v_5\}$ with interval representation $V_c = \{[4, 6), [5, 8), [7, 9), [2, 5), [0, 3)\}$ and $m_c = 2$ pages of fixed size. Basic block v_1 conflicts with v_2, v_4, for example, and cannot occupy the same page otherwise they would attempt to store data at the same addresses in virtual memory. One possible placement consists of slotting v_1, v_3, v_5 into Page 1 and slotting v_2, v_4 into Page 2, as v_2, v_4 do not conflict and v_1, v_3, v_5 do not mutually conflict.

Formally, given intervals $\{I_v\}_{v \in V}$ on a line, the *interval graph* $G = (V, E)$ has $(u, v) \in E$ iff $I_u \cap I_v \neq \emptyset$ [34, 21]. We thus define the interval graph $G_c = (V_c, E_c)$ where $(i, j) \in E_c \iff I_i \cap I_j \neq \emptyset$ for basic blocks v_i, v_j and G_c is an interval graph by construction.

Then, a *vertex labeling* of a graph $G = (V, E)$ with k vertex labels is a function $f : V \to \{1, \ldots, k\}$ assigning a vertex label to each vertex. A vertex labeling is a *proper vertex labeling* if $(u, v) \in E \implies f(u) \neq f(v)$, or in other words, no two vertices sharing an edge share the same label. Thus, a valid assignment of our basic blocks to pages is a function $f : V_c \to [\, m_c \,]$ such that $(i, j) \in E_c \implies f(i) \neq f(j)$, and thus, a proper vertex labeling of G_c using m_c labels. The *chromatic polynomial* χ_G of a graph counts the number of these proper vertex labelings as a function of the number of available vertex labels [41]. It is our goal to find χ_{G_c}, the counting function for the vertex-labelings, and thus, the counting function for possible randomization layouts.

Example (cont'd). If we construct the graph with vertices v_1, v_2, v_3, v_4, v_5, with edges between any 2 vertices whose corresponding basic blocks conflict, we find that our earlier placement of v_1, v_3, v_5 into Page 1 and v_2, v_4 into Page 2 corresponds to a vertex labeling of the interval graph with vertices v_1, v_3, v_5 of one label and v_2, v_4 a second label.

For graphs containing a cycle, a *chord* is an edge that is not part of the cycle, but connects two vertices of that cycle. A graph is *chordal* if every cycle of length at least four has a chord. Call the set of neighbors of a vertex v_i the set $N(v_i) = \{v_j \in V \mid (v_i, v_j) \in E\}$, *i.e.,* the set of vertices adjacent to v_i. A vertex ordering $v_1, \ldots, v_n$ is a *perfect elimination ordering (PEO)*, sometimes called a *simplicial elimination ordering*, if for every i, the neighbors appearing after v_i in the order, $N^+(v_i) = N(v_i) \cap \{v_{i+1} \ldots v_n\}$, form a clique (*i.e.,* each pair of vertices of the clique has an edge) [42]. A graph is chordal if and only if it admits a PEO [21, 18]. Since interval graphs are a known subclass of chordal graphs, G_c must admit a PEO.

▶ **Definition 1** (Right-endpoint ordering). *Let $G = (V, E)$ be an interval graph with an interval representation $I_v = [\ell_v, r_v)_{v \in V}$. A* right-endpoint ordering *of G is any vertex ordering $v_1, \ldots, v_n$ such that $r_{v_1} \leq r_{v_2} \leq \ldots r_{v_n}$ where vertices are ordered by nondecreasing right-endpoint and ties may be broken arbitrarily.*

▶ **Lemma 2.** *Let G be an interval graph with interval representation $I_v = [\ell_v, r_v)$. Any right-endpoint ordering of G (Definition 1) is a perfect elimination ordering. In particular, the right-endpoint ordering of G_c over the half-open intervals $I_{v v \in V_c}$ of basic blocks is a PEO.*

Proof. Fix i and let $v_j, v_k \in N^+(v_i)$ with $j, k > i$. Since v_j overlaps v_i and $r_{v_j} \geq r_{v_i}$, we have $\ell_{v_j} < r_{v_i}$ and similarly $\ell_{v_k} < r_{v_i}$ and $r_{v_k} \geq r_{v_i}$. Thus we have that $\max(\ell_{v_j}, \ell_{v_k}) < r_{v_i} \leq \min(r_{v_j}, r_{v_k})$, so $I_{v_j} \cap I_{v_k} \supseteq [\max(\ell_{v_j}, \ell_{v_k}), r_{v_i}) \neq \emptyset$. Thus v_j and v_k are adjacent, and $N^+(v_i)$ is a clique. ◀

Unlike general graphs, chordal graphs admit a closed-form expression for the number of proper labelings, due to how PEOs reveal constraints for neighbors in a clique. The next lemma is a well-known consequence of simplicial elimination in chordal graphs.

▶ **Lemma 3** (Remark 2.5 in [2]). *Given a perfect elimination ordering, $v_1, \ldots, v_n$ of a chordal graph G with v a simplicial vertex of degree d. In a proper k-labeling of G, the neighbors of v have d different labels and leave $k - d$ remaining labels to choose from. Thus, by the product rule for counting the ways to label the vertices in the ordering $v_1 \ldots v_n$,*

$$\chi_G = \prod_{i=1}^{n} (k - d_i)$$

▶ **Theorem 4.** *Let $v_1 \ldots v_{n_c}$ be vertices of interval graph G_c ordered in the right-endpoint ordering (Definition 1). Let $q_{c,i} = |N^+(v_i)|$ in this ordering. Then the number of valid placements is*

$$N_c = \prod_{i=1}^{n_c} (m_c - q_{c,i})$$

Proof. By Lemma 2, $v_1, \ldots, v_{n_c}$ is a PEO of G_c. We then count the proper labelings of G_c with the available labels $[m_c]$ by labeling in reverse order as $v_{n_c}, v_{n_c - 1}, \ldots v_1$. When labeling v_i the neighbors $N^+(v_i)$ are already labeled. Because $N^+(v_i)$ is a clique, those neighbors must all have distinct labels, so exactly $q_{c,i}$ vertex labels have been used, and there are $(m_c - q_{c,i})$ label choices remaining for v_i. Thus, by the product formulation of Lemma 3, $\chi_{G_c}(m_c) = \prod_{i=1}^{n_c} (m_c - q_{c,i})$ is the number of proper vertex labelings of G_c in this form and thus the number of possible basic block placements into m_c pages. ◀

Thus, we have the solution for basic-block placements into pages of a single page color. However, since basic blocks can only be placed into pages of the same color, each color forms an independent problem, and so we have that the total number of combinations is,

$$N = \prod^{c} N_c = \prod^{c} \prod_{i=1}^{n_c} (m_c - q_{c,i})$$

Finally, since layouts are sampled uniformly from the total valid placements as counted above, the Shannon entropy of a particular layout choice in bits is,

$$H = \log_2(N) = \log_2(\prod^{c} \prod_{i=1}^{n_c} (m_c - q_{c,i})) = \sum_{c} \sum_{i=1}^{n_c} \log_2(m_c - q_{c,i})$$

indicating how likely a particular ordering is to occur and, correspondingly, how likely an adversary is to be able to predict a particular BBL-page ordering.

Example (cont'd). For the graph consisting of v_1, v_2, v_3, v_4, v_5, we find that there are 2 proper vertex-labelings of the interval graph. Starting with an unlabeled graph, we order the basic blocks by nondecreasing right endpoint as v_5, v_4, v_1, v_2, v_3, producing our desired perfect elimination ordering. In reverse order, we label the graph in the sequence v_3, v_2, v_1, v_4, v_5. Thus, for v_3, since no neighbors have yet been labeled, $(2 - 0)$ possible label choices exist. Next, v_2 is in a clique with forward neighbor v_3 and thus it has $(2 - 1)$ possible labels. Repeating for the whole ordering, we construct $N = (2 - 0)(2 - 1)(2 - 1)(2 - 1)(2 - 1) = 2$ possible vertex labelings of the interval graph matching the corresponding 2 possible placements of the basic blocks into 2 pages.

6.2 Entropy Evaluation

Figures 8 and 9 report the Shannon entropy H induced by DART's page-level randomization for each TACLeBench binary, where N is the count of valid basic-block-to-page placements per the closed-form formula. This entropy measures the size of the layout search space given the constraints for each binary. Due to uniform sampling over the N valid layouts, an attacker attempting to guess a predicted layout succeeds with probability 2^{-H}. Additionally, compared to traditional ASLR, where partial pointer overwriting allows for overwriting only the known non-randomized bits of a pointer, DART's model of per-page randomization ensures this is probabilistically difficult as modeled by the entropy H. We note that the comparison with techniques such as AES-128 in the figures is not a claim of cryptographic security, but rather a comparison of combinatorial search space at known scales.

At $m_c = 16$ pages per color (Figure 8), all binaries remain above classically brute-forceable ranges, and outperform standard Address Space Layout Randomization, which for 64-bit Linux randomizes 28 bits of every pointer. Additionally, most binaries exhibit hundreds to thousands of bits of layout entropy, and for $m_c = 64$ pages per color (Figure 8), all binaries cross 100 bits of entropy and can easily be considered "strongly secure". Larger binaries, such as zip-test (2,649 basic blocks), have entropy as high as $\approx 10^4$ despite having overlapping basic block intervals, which the smaller binaries often do not.

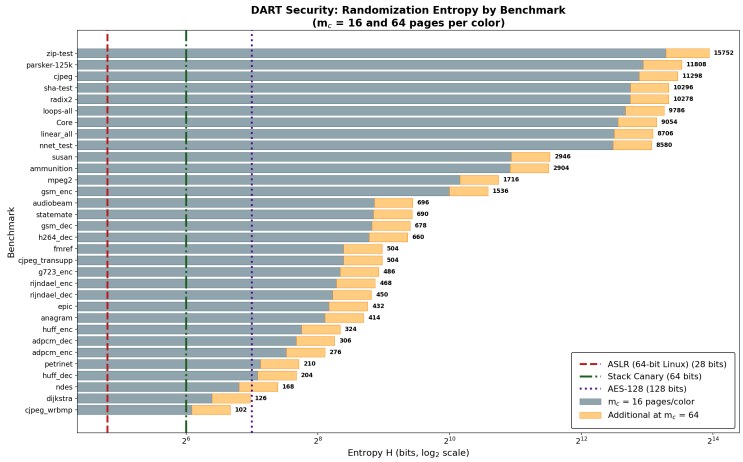


Figure 8 DART Security of TACLeBench Suite compared against security of known measures (i.e. ASLR, stack canaries, AES-128) with 16 and 64 pages per color.

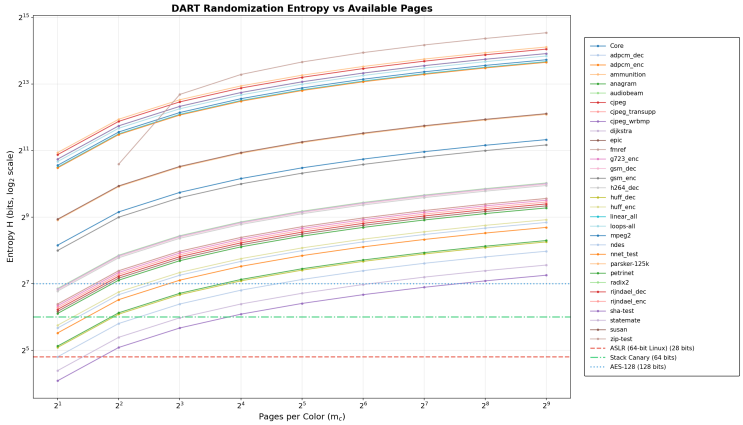


Figure 9 DART Randomization Entropy vs Available Pages Per Color (TACLeBench Suite).

Figure 9 shows H as a function of m_c and how, despite few pages per color available, DART induces entropy greater than that of standard security measures. The linearity of many benchmarks can be noted as arising from few or no conflicting basic-block intervals, causing the closed-form formula to collapse to $H(m_c) \approx \sum_c \log_2(m_c)$ for small binaries. Deviations from perfect linearity and varying slopes showcase how m_c being close to local clique sizes (*i.e.*, $q_{c,i}$) induces constraints, but the entropy still far outperforms known measures.

7 Related Work

Randomization defenses date back to the PaX team development of address space layout randomization (ASLR) in 2003 [49] (see [52] for a review of randomization defenses). ASLR randomizes at the segment level, moving each segment as a whole. It requires position-independent code (PIC), but is otherwise implemented purely in the OS via the virtual-to-physical mapping. While this is simple and efficient, studies have shown that this coarse-grained randomization is vulnerable if attacks can leak code pointers, as they can

therefore determine the relative location of other vulnerable code [54]. Nonetheless, ASLR raises the bar for attackers with minimal overhead, and is applied nearly ubiquitously in general-purpose systems.

More recent randomization defenses have provided sub-segment randomization via compiler instrumentation. For example, selfrando [14] randomizes at the function-level, and compiler-assisted code randomization (CCR) permutes at either the function- or BBL-level [29]. Instruction-level randomization changes the offsets at the function level by inserting no-ops between instructions [27, 25], and is the highest precision of randomization, though BBLs are the lowest level at which instructions can be re-ordered at scale. Another area of research has been exploring the frequency of randomization; other techniques have changed this from simply once at compile time to dynamically at runtime [5].

Within the real-time systems community, randomization is widely viewed as incompatible with timing predictability and verification. The effects of some randomization defenses have been studied by Burow *et al.* [7] and shown to increase non-determinism, and Fellmuth *et al.* [19] found cache-nondeterminism from randomization can affect WCET analysis and reported WCET bounds as high as 1.8× those of undiversified binaries. STABILIZER [16] repeatedly randomizes code, stack, and heap layout at runtime, forcing layout-induced timing into a Gaussian distribution for statistical analysis. These works showcase the effects DART addresses and share the same underlying diagnosis, however DART attempts to make layout-induced timing effects attributable to the binary itself. There have been recent studies of randomization in embedded systems, especially microcontroller-based. For example, limited randomization in the EPOXY work [12] applies randomization at the physical level in microcontrollers without virtual memory. Continuous re-randomization has also been developed for microcontrollers [46].

A similar line of work uses randomization for inducing randomized cache replacement [30] or for random code and data placement [10, 31] along with analysis techniques using extreme-value theory for probabilistic WCET bounds. Cros *et al.* [15] showcase how dynamic randomization introduces runtime indirections and pointer-dense code that makes functional verification of the resulting binary particularly difficult. TASA [32] builds on this by developing software randomization with only source-to-source transformations and no modifications of compiler, linker, or runtime behavior allowing more effective measurement-based probabilistic timing analysis. Together with DART, these works form a path where STABILIZER and Cros *et al.* randomize to normalize and sample timing variation from layouts, TASA randomizes at the source level to avoid toolchain cost, and DART randomizes to fix per-block color and offsets so cache behavior becomes a property of the binary rather than of OS allocator state. All works are responses to the same underlying observation that timing variance under randomization is driven by layout-induced cache behavior.

Cache management. Shared caches in multicore processors are a significant source of non-determinism, as cores can evict data reused by others. Thus, there has been significant work on cache management in the real-time community to address or eliminate this (*e.g.,* [28, 36, 53, 55]). These techniques rely on cache coloring to control which tasks or cores may access certain colors or regions of the cache. We refer the interested reader to [23].

Our work has two important distinctions from previous work on cache management. First, prior work largely focused on the data cache, or did not differentiate instructions and data (*i.e.,* allocating both in the same pool of pages otherwise isolated from other cores). Second, real-time cache-management approaches have sought to mitigate inter-task cache contention while we study the effects of instruction-level intra-task non-determinism, asking how randomizing the physical address space impacts performance and whether intra-task interference can be measured and controlled.

8 Conclusion

We have presented DART, a real-time address randomization defense providing strong layout uncertainty for code segments in virtual memory whilst making cache color and alignment statically determinable from the binary, so that layout-induced timing variability is bounded across the space of layouts the defense produces and amenable to build-time worst-case timing analysis. DART is a full-chain framework that combines modifications to the compiler toolchain, binary and linker metadata, and OS kernel support to fully realize these cache-color and alignment constraints during runtime. We evaluate effects on performance and present a graph-theoretic derivation of the randomization entropy DART provides. With monochrome providing a measurement-derived bound on the coloring-conflict component and striped tracking baseline timing on cache-tolerant binaries, DART is positioned as a suitable defense for real-time and embedded environments with need for strong real-time and cache-behavior guarantees.

References

1 Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

2 Geir Agnarsson. On chordal graphs and their chromatic polynomials. *MATHEMATICA SCANDINAVICA*, 93(2):240–246, December 2003. doi:10.7146/math.scand.a-14421.

3 Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the mirai botnet. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1093–1110. USENIX Association, August 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis.

4 Cl'ement Ballabriga, Hugues Cass'e, Christine Rochange, and Pascal Sainrat. OTAWA: An open toolbox for adaptive WCET analysis. In *IFIP Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS)*, pages 35–46. Springer, 2010. doi:10.1007/978-3-642-16256-5_6.

5 David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. Timely rerandomization for mitigating memory disclosures. In *ACM Conference on Computer and Communications Security*, CCS, 2015.

6 Sergey Bratus, Michael E Locasto, Meredith L Patterson, Len Sassaman, and Anna Shubina. Exploit programming: From buffer overflows to weird machines and theory of computation. *USENIX; login*, 36(6):13–21, 2011. URL: https://www.usenix.org/publications/login/december-2011-volume-36-number-6/exploit-programming-buffer-overflows-weird.

7 Nathan Burow, Ryan Burrow, Roger Khazan, Howard Shrobe, and Bryan C Ward. Moving target defense considerations in real-time safety-and mission-critical systems. In *Proceedings of the 7th ACM Workshop on Moving Target Defense*, pages 81–89, 2020. doi:10.1145/3411496.3421224.

8 Nicholas Carlini and David Wagner. ROP is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium*, USENIX Sec, 2014.

9 Francisco J. Cazorla, Leonidas Kosmidis, Enrico Mezzetti, Carles Hernandez, Jaume Abella, and Tullio Vardanega. Probabilistic worst-case timing analysis: Taxonomy and comprehensive survey. *ACM Computing Surveys*, 52(1):14:1–14:35, 2019. doi:10.1145/3301283.

10 Francisco J. Cazorla, Eduardo Qui nones, Tullio Vardanega, Liliana Cucu, Benoit Triquet, Guillem Bernat, Emery Berger, Jaume Abella, Franck Wartel, Michael Houston, Luca Santinelli, Leonidas Kosmidis, Code Lo, and Dorin Maxim. PROARTIS: Probabilistically analyzable real-time systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2s):94:1–94:26, 2013. doi:10.1145/2465787.2465796.

11 S. Checkoway, L. Davi, A. Dmitrienko, A.R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *ACM Conference on Computer and Communications Security*, CCS, 2010.

12 A. Clements, N. Almakhdhub, K. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer. Protecting bare-metal embedded systems with privilege overlays. In *S&P '17*, 2017.

13 TIS Committee et al. Tool interface standard (tis) executable and linking format (elf) specification version 1.2, 1995.

14 Mauro Conti, Stephen Crane, Tommaso Frassetto, Andrei Homescu, Georg Koppen, Per Larsen, Christopher Liebchen, Mike Perry, and Ahmad-Reza Sadeghi. Selfrando: Securing the tor browser against de-anonymization exploits. *Proceedings on Privacy Enhancing Technologies*, 2016(4):454–469, 2016. doi:10.1515/POPETS-2016-0050.

15 Fabrice Cros, Leonidas Kosmidis, Franck Wartel, David Morales, Jaume Abella, Ian Broster, and Francisco J. Cazorla. Dynamic software randomisation: Lessons learned from an aerospace case study. In *Design, Automation & Test in Europe Conference (DATE)*, pages 103–108, 2017. doi:10.23919/DATE.2017.7926966.

16 Charlie Curtsinger and Emery D. Berger. STABILIZER: Statistically sound performance evaluation. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2013. doi:10.1145/2451116.2451141.

17 Alessandro Di Pinto, Younes Dragoni, and Andrea Carcano. TRITON: The first ICS cyber attack on safety instrument systems. In *Proc. Black Hat USA*, pages 1–26, 2018.

18 G. A. Dirac. On rigid circuit graphs. *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, 25(1-2):71–76, 1961.

19 Joachim Fellmuth, Thomas Göthel, and Sabine Glesner. Instruction caches in static WCET analysis of artificially diversified software. In *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.ECRTS.2018.21.

20 Christian Ferdinand and Reinhold Heckmann. ait: Worst-case execution time prediction by static program analysis. In *Building the Information Society*, pages 377–383. Springer, 2004. doi:10.1007/978-1-4020-8157-6_29.

21 Delbert Ray Fulkerson and Oliver A. Gross. Incidence matrices and interval graphs. *Pacific Journal of Mathematics*, 15(3):835–855, 1965. doi:10.2140/pjm.1965.15.835.

22 Dan Goodin. That time a patient's heart procedure was interrupted by a virus scan. Ars Technica, May 2016. URL: https://arstechnica.com/information-technology/2016/05/that-time-a-patients-heart-procedure-was-interrupted-by-a-virus-scan/.

23 Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. A survey on cache management mechanisms for real-time embedded systems. *ACM Computing Surveys (CSUR)*, 48(2):1–36, 2015. doi:10.1145/2830555.

24 Enes Göktas, Benjamin Kollenda, Philipp Koppe, Erik Bosman, Georgios Portokalidis, Thorsten Holz, Herbert Bos, and Cristiano Giuffrida. Position-independent code reuse: On the effectiveness of aslr in the absence of information disclosure. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 227–242, 2018. doi:10.1109/EuroSP.2018.00024.

25 J. Hiser, A. Nguyen, M. Co, M. Hall, and J.W. Davidson. ILR: Where'd my gadgets go. In *33rd IEEE Symposium on Security and Privacy*, S&P, 2012.

26 Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. Micro-gadgets: size does matter in turing-complete return-oriented programming. In *Proceedings of the 6th USENIX conference on Offensive Technologies*, pages 7–7. USENIX Association, 2012.

27 Todd Jackson, Babak Salamat, Andrei Homescu, Karthikeyan Manivannan, Gregor Wagner, Andreas Gal, Stefan Brunthaler, Christian Wimmer, and Michael Franz. Compiler-generated software diversity. In *Moving Target Defense*, pages 77–98. Springer, 2011. doi:10.1007/978-1-4614-0977-9_4.

28 Namhoon Kim, Bryan C Ward, Micaiah Chisholm, James H Anderson, and F Donelson Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. *Real-Time Systems*, 53(5):709–759, 2017. doi:10.1007/S11241-017-9272-9.

29 Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P Kemerlis, and Michalis Polychronakis. Compiler-assisted code randomization. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 461–477. IEEE, 2018. doi:10.1109/SP.2018.00029.

30 Leonidas Kosmidis, Jaume Abella, Eduardo Qui nones, and Francisco J. Cazorla. A cache design for probabilistically analysable real-time systems. In *Design, Automation & Test in Europe Conference (DATE)*, pages 513–518, 2013.

31 Leonidas Kosmidis, Charlie Curtsinger, Eduardo Qui nones, Jaume Abella, Emery D. Berger, and Francisco J. Cazorla. Probabilistic timing analysis on conventional cache designs. In *Design, Automation & Test in Europe Conference (DATE)*, pages 603–606, 2013.

32 Leonidas Kosmidis, Roberto Vargas, David Morales, Eduardo Qui nones, Jaume Abella, and Francisco J. Cazorla. TASA: Toolchain-agnostic static software randomisation for critical real-time systems. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. ACM, 2016.

33 Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy*, pages 276–291. IEEE, 2014. doi:10.1109/SP.2014.25.

34 C. Lekkeikerker and J. Boland. Representation of a finite graph by a set of intervals on the real line. *Fundamenta Mathematicae*, 51(1):45–64, 1962. URL: http://eudml.org/doc/213681.

35 Bill Mahoney, Philip Sigillito, Jeff Smolinski, Todd McDonald, and George Grispos. Analyzing the performance of block-splitting in llvm fingerprinting. In *International Conference on Cyber Warfare and Security*, pages 176–184. ACI, 2022. doi:10.34190/iccws.17.1.39.

36 Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54. IEEE, 2013. doi:10.1109/RTAS.2013.6531078.

37 Microsoft. A detailed description of the data execution prevention (dep) feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003. Online, September 2006. URL: http://support.microsoft.com/kb/875352/en-us.

38 Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015.

39 Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *ACM Conference on Programming Language Design and Implementation*, PLDI, 2009.

40 OpenBSD. Openbsd 3.3, 2003.

41 Ronald C. Read. An introduction to chromatic polynomials. *Journal of Combinatorial Theory*, 4(1):52–71, 1968. doi:10.1016/S0021-9800(68)80087-0.

42 Donald J. Rose, R. Endre Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5(2):266–283, 1976. doi:10.1137/0205021.

43 Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *36th IEEE Symposium on Security and Privacy*, S&P, 2015.

44 Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

45 Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security*, CCS, 2007.

46 Jiameng Shi, Le Guan, Wenqiang Li, Dayou Zhang, Ping Chen, and Ning Zhang. HARM: Hardware-assisted continuous re-randomization for microcontrollers. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 520–536. IEEE, 2022. doi:10.1109/EUROSP53844.2022.00039.

47 Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. Sok: sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295. IEEE, 2019. doi:10.1109/SP.2019.00010.

48 Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proc. of IEEE Symposium on Security and Privacy*, 2013.

49 PaX Team. PaX address space layout randomization (aslr), 2003. URL: ttp://pax.grsecurity.net/docs/aslr.txt.

50 United States Department of Homeland Security. ICS advisory (ICSA-21-119-04), 2021. URL: https://us-cert.cisa.gov/ics/advisories/icsa-21-119-04.

51 Viasat, Inc. Ka-sat network cyber attack overview. https://www.viasat.com/perspectives/corporate/2022/ka-sat-network-cyber-attack-overview/, 2022. Accessed: 2025-03-30.

52 Bryan C. Ward, Steven R. Gomez, Richard W. Skowyra, D. Bigelow, Jason N. Martin, James W. Landry, and Hamed Okhravi. Survey of cyber moving targets second edition. Technical Report 1228, MIT Lincoln Laboratory, January 2018.

53 Bryan C Ward, Jonathan L Herman, Christopher J Kenna, and James H Anderson. Making shared caches more predictable on multicore platforms. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 157–167. IEEE, 2013.

54 Bryan C Ward, Richard Skowyra, Chad Spensky, Jason Martin, and Hamed Okhravi. The leakage-resilience dilemma. In *European Symposium on Research in Computer Security*, pages 87–106. Springer, 2019. doi:10.1007/978-3-030-29959-0_5.

55 Meng Xu, Linh Thi, Xuan Phan, Hyon-Young Choi, and Insup Lee. vCAT: Dynamic cache management using cat virtualization. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 211–222. IEEE, 2017.

56 Ruotong Yu, Francesca Del Nin, Yuchen Zhang, Shan Huang, Pallavi Kaliyar, Sarah Zakto, Mauro Conti, Georgios Portokalidis, and Jun Xu. Building embedded systems like it's 1996. *Network and Distributed Systems Security (NDSS) Symposium*, 2022.

Distribution Statement

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited. This material is based upon work supported by the Under Secretary of War for Research and Engineering under Air Force Contract No. FA8702-15-D-0001 or FA8702-25-D-B002. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of War for Research and Engineering. © 2026 Massachusetts Institute of Technology. Delivered to the U.S. Government with Unlimited Rights, as defined in DFARS Part 252.227-7013 or 7014 (Feb 2014). Notwithstanding any copyright notice, U.S. Government rights in this work are defined by DFARS 252.227-7013 or DFARS 252.227-7014 as detailed above. Use of this work other than as specifically authorized by the U.S. Government may violate any copyrights that exist in this work.