# DAPPER: A Lightweight and Extensible Framework for Live Program State Rewriting

Abhishek Bapat*
*Virginia Tech*
Blacksburg, USA
abapat28@vt.edu

Jaidev Shastri*
*Virginia Tech*
Blacksburg, USA
jaidevshastri@vt.edu

Xiaoguang Wang*
*University of Illinois Chicago*
Chicago, USA
xgwang9@uic.edu

Abilesh Sundarasamy
*Virginia Tech*
Blacksburg, USA
abisundar@vt.edu

Binoy Ravindran
*Virginia Tech*
Blacksburg, USA
binoy@vt.edu

***Abstract*—**
**We present DAPPER, a lightweight system that transforms the execution state of a live process into a new process state in a secure and extensible manner. DAPPER checkpoints a live process into a process image using Linux's CRIU mechanism, rewrites the image with an updated execution state, and restores program execution. In particular, DAPPER can restore the program execution on a CPU with a different architecture by rewriting the process's architecture-specific execution state. DAPPER transforms the process externally and only requires inserting a small amount of compile-time metadata to guide the state transformation. Therefore, DAPPER brings a smaller attack surface for the transformed program and can be extended for different scenarios in contrast to existing techniques. We build and evaluate a prototype of DAPPER using server applications and benchmark suites. Our evaluation shows that DAPPER can be extended and used in many different scenarios, such as improving servers' energy efficiency by live program migration on heterogeneous processors and enhancing program security with dynamic randomness of the program states.**

***Index Terms*—Process state rewriting, heterogeneous processors, live migration**

## I. INTRODUCTION

Runtime software state transformation and relocation are used in several real-world scenarios. Many research efforts transform and relocate a running program for live migration [11], [50], [57], dynamic software patching [21], [34], and program layout re-randomization [7], [59], to name a few. For example, live process or virtual machine (VM) migration supports more efficient load balancing in the data center environment and improves infrastructure maintainability. A typical live migration system transfers the application's memory, storage, and network connectivity from the original machine to the destination [57]. The migrated program keeps the same program state as before live migration.

Recently, there is an emerging trend of ISA-heterogeneity in cloud systems and high-end servers. For example, Amazon [63] and Oracle [41] recently introduced ARM-based servers in their cloud platforms, which are traditionally made up of x86-based servers, to provide low-cost computing capabilities. Emerging high-end I/O devices for x86-based servers such as SmartNICs [36], [37] and SmartSSDs [35], [46] include full-featured SoCs with ARM-based CPUs. These works use different ISAs to improve resource utilization by dynamically relocating computation to a more suitable architecture [5], [40], [48]. For example, HIPStR [47] dynamically migrates program execution across ISA-different CPUs to increase entropy; HEXO [40] offloads unikernelized computing workloads to embedded devices to save cost and reduce energy usage. These works use complex software stacks. For example, HIPStR combines the native compilation and dynamic binary translation to convert register states across ISAs [47]. Such *complex* software subsystems add a burden to software deployment and also increase the attack surface.

This paper presents DAPPER, a *lightweight framework* that dynamically updates the architectural and program state of running processes. DAPPER enables transforming the program state from an original process to a new one to adapt to the changing and diversified hardware/software environment. DAPPER utilizes Linux's CRIU mechanism [15] for checkpointing and restoring a process. DAPPER takes a CRIU snapshot and rewrites the snapshot to update the program execution state according to a user-defined policy. Example policies include changing the program state (*i.e.,* registers and memory) so that the process is restored to a different architecture, or periodically re-randomizing the function call stack by changing the layout of each function stack frame. Other possible policies can be live software updates [21] or dynamic software feature customization [25], to name a few.

We build a prototype of DAPPER and evaluate it using real-world applications, such as the Nginx web server, the Redis key/value store and benchmark suites, including the NAS Parallel Benchmark (NPB) suites [61], the Linpack benchmark [56] and the PARSEC benchmark [6]. We demonstrate that DAPPER can dynamically rewrite the process state to relocate the process on CPUs of different architectures[1]. Our evaluation shows that DAPPER can dynamically relocate running programs across machines of different ISAs in 600 *ms* to 2.3 seconds, depending on the application's memory footprint size at the checkpoint. Using this cross-architecture

* A. Bapat, J Shastri, and X. Wang contributed equally to this work.



[1] A demo can be found at https://github.com/dapper-project/demo.

process rewriting capability, DAPPER can improve the energy efficiency for HPC workloads by up to 39% when dynamically evicting jobs to low-power Raspberry Pi boards. We also show that DAPPER allows periodic re-randomizing of the function call stack, defeating several classes of CVEs [13], [24], [42].

The paper's contributions include:

- We propose a process rewriting mechanism for runtime architectural and program state transformation;
- We present the design and implementation of DAPPER, a modular and extensible framework for live updating the runtime program state; DAPPER will be open-sourced after publication: https://github.com/dapper-project/dapper;
- We evaluate DAPPER 's effectiveness and efficiency using real-world applications and benchmark suites and show that DAPPER can be used in multiple scenarios, such as consolidating workload on heterogeneous CPUs and improving software runtime security.

## II. BACKGROUND AND MOTIVATION

Runtime program state rewriting allows changing the internal states of a program at runtime. Examples of such systems allow the dynamic change of the address space layout [7], [59], updating register values and the architecture state [47], [48], [52], dynamic binary translation (DBT) [51] or dynamic software update (DSU) [21], [34], among others. There is a trade-off between the complexity of the system and the functionalities these systems can provide. Some existing approaches heavily rely on a complex software stack, such as the modified operating system kernels or system/architecture simulators [5], [47], to obtain and convert the runtime program state. The complication of such systems burdens software deployment, preventing them from being deployed in a real-world environment. On the other hand, systems that transform program states in the user-space [7], [34], [59] may not extend themselves to a more complicated task at the OS or architecture level. In contrast, DAPPER is lightweight and extensible (Fig. 1). DAPPER was built on top of existing and widely used software stacks. The software stack modification is minimal: we mainly extended a CRIU image examination tool [16] to rewrite the process image, leaving the core software stack untouched; yet, DAPPER can rewrite program states for different purposes.

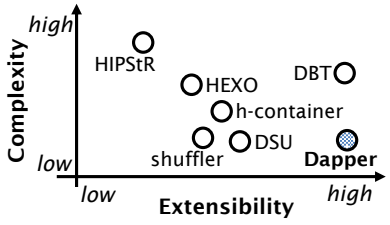


Fig. 1. A comparison of DAPPER to other competitor techniques in complexity and extensibility.

Live migration [57] is a technique that moves application instances (*i.e.,* processes or containers) across different machines without disconnecting networked clients. The technique has primarily been used for fault-management, server maintenance (*e.g.,* OS kernel update), and load balancing in datacenters. CRIU [15] is a Linux mechanism that supports process/container live migration. To migrate a process, CRIU dumps a process's state into a set of image files, which are then transferred over a network connection to a destination machine. These files are then used to restore the process's execution on the destination machine. Most CRIU-generated images are JSON objects encoded as byte vectors using Google's protocol buffer format (`protobuf`) [23]. The exceptions are raw image files that contain the process's memory snapshot at the checkpoint. CRIU also includes a tool called CRIT [16], which can be used to examine the process images in the `protobuf` format, decode them to human-readable JSON files (`decode`), and encode them back to `protobuf` (`encode`). DAPPER extends CRIU/CRIT to rewrite process images for different types of dynamic software transformation.

## III. SYSTEM DESIGN AND IMPLEMENTATION

Fig. 2 shows DAPPER's high-level overview. Similar to most existing dynamic program state transformation systems [5], [7], [10], [34], [52], [59], DAPPER also requires inserting (minimal) metadata into the program binary to guide runtime state transformation (Section III-A). Once the program is instrumented with the metadata and is spawned as a process, DAPPER utilizes the *checkpoint and restore mechanism* to suspend the process, rewrite the process state, and restore the process execution (Section III-B). The transformed process can continue executing on a different (or the same) machine, depending on the user-defined transformation policy and the requirement. DAPPER can also transform the process images into a new process that can be restored to run on a different architecture while retaining the same program functionalities and execution context (Section III-C).

DAPPER allows end-users to define different transformation policies. For example, end-users can rewrite the architecture state to enable cross-architecture process migration to consolidate heterogeneous computing resource utilization [40]. They can also shuffle the internal program state to implement a code/data re-randomization system [7], [10], [59]. DAPPER allows end-users to control when the transformation happens.

### A. Metadata for Runtime Process Transformation

DAPPER transforms a program at runtime by rewriting the process image. Similar to existing approaches [7], [10], [31], [52], [59], DAPPER requires end-users to specify the transformation policy and insert metadata into the binary. We leverage metadata generated from compiler toolchains such as debugging information (*i.e.,* DWARF [12]) and live value records (*i.e.,* LLVM stack maps [29]) to instrument the binary. DAPPER uses this metadata to locate code and live values for runtime program state transformation.

To support cross-architectural state transformation, DAPPER requires the target process to be paused at architecture-agnostic *equivalence points* [49]. An equivalence point allows program

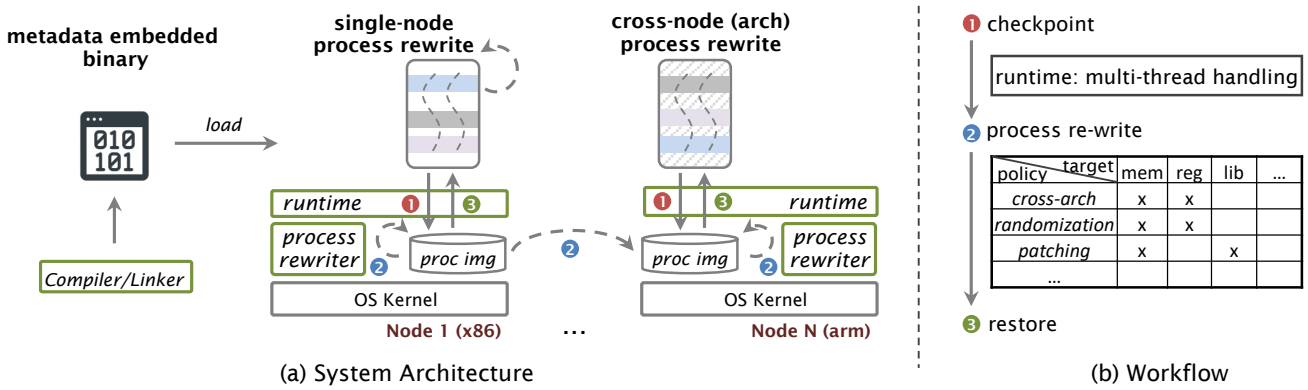


Fig. 2. DAPPER's system architecture and workflow. The DAPPER runtime controls the process execution; the DAPPER process rewriter transforms the process image with different transformation policies.

states (*e.g.,* register values, stack frames) to be safely converted across different architectures. Similar to cross-architecture dynamic binary translation (DBT) techniques [51], instructions and register values can only be transformed at basic block and function boundaries. In compiler design, an architecture-independent instruction (*e.g.,* an LLVM intermediate representation or IR [27]) is typically translated to several machine instructions, necessitated in part because an unlimited number of virtual registers in the IR form must be mapped to a fixed number of physical registers available for the target machine. Since this mapping is architecture-dependent and the compiler backend may produce different machine instruction sequence for different architectures, the program state may not have consistent memory semantics across different architectures at an arbitrary machine instruction besides the boundaries of basic blocks and functions [48]. In current design, DAPPER considers function boundaries as equivalence points and suspends the target process only at these locations.

Other program state transformation systems may not require such restrictions for suspending the process. For example, a code/stack re-randomization system may pause the process at arbitrary locations but keep the current function call context unchanged [59]. We demonstrate the process-level program state rewriting capability by extending DAPPER for a stack re-randomization system. In particular, we reuse the metadata inserted for architectural state transformation (*i.e.,* the stack map [29]) to shuffle the stack layout. In compiler design, a stack map records the location of *live values* at an instruction address [58]. The live values indicate the program variables that have downstream uses. Therefore, the live values give us enough information for transforming the valid program variables (typically stored in registers and on the call stack) upon resuming program execution. The live values are architecture-independent, as they are calculated and generated by the compiler's middle-end passes [29].

We also instrument an LLVM intrinsic function at each equivalence point to generate the stack map records. An LLVM intrinsic is a built-in function in the compiler backend that takes advantage of hardware capabilities for performing specialized operations [28]. We extend the LLVM compiler to automatically instrument the function boundary and add the LLVM intrinsic function to generate the stack map into an ELF section. DAPPER's process rewriter takes advantage of the stack map section in the binary to transform the architectural state and stack layout (Section III-C). Although we currently only rewrite the stack object locations and the architectural state, we believe that DAPPER can also support other program state transformation strategies, such as dynamic software update and dynamic code customization.

### B. *The* DAPPER *Runtime*

After the program is instrumented with metadata, we launch the application under the DAPPER runtime. The DAPPER runtime gives end-users control over how and when to apply transformation policies to the target process. As previously mentioned, we pause the process and rewrite it only at equivalence points. A straightforward approach for pausing the process is to insert a debug instruction[2] at an equivalence point ahead of the program execution counter. This is similar to how a debugger pauses a process for debugging purposes. However, dynamically inserting the debugging instruction brings potential synchronization challenges: it is difficult to know where is the next equivalence point given a program counter. Moreover, the signal generated by a breakpoint instruction will pause all threads at the same time [54]. As a result, there is no guarantee for other threads to stop at equivalence points, which will likely cause a non-transformable state for cross-architecture state transformation. To solve this problem, DAPPER instruments a simple *checker function* for each equivalence point and collaborates with a multi-threaded code monitor to pause and transform the program into a transformable state.

Fig. 3 illustrates how the DAPPER runtime pauses a multi-threading program. Each equivalence point is instrumented with a *checker function* that waits for the signal to transform the process. When the runtime raises a signal for cross-architecture transformation, the checker function raises

[2]Examples include the `int3` instruction (`0xCC`) on the x86 architecture and the instruction of bytes `0xD4200000` on the ARM64 architecture.

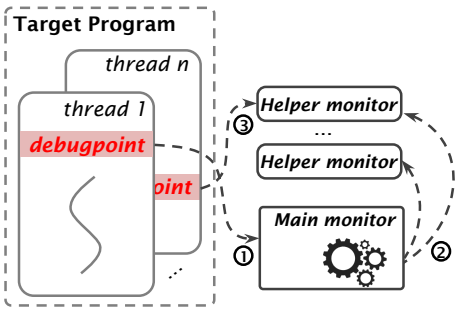


Fig. 3. DAPPER's runtime monitor for interrupting multi-threading applications.

a `SIGTRAP` exception by touching a breakpoint instruction (① in Fig. 3). Again, simply pausing the whole process will cause other threads to stop at wrong places. DAPPER solves this problem by creating multiple *helper monitors* and dynamically attaching a helper monitor (implemented with `ptrace` [33]) to each thread (② in Fig. 3). The Linux `ptrace` interface allows attaching each thread to potentially different tracers [33]. Therefore, even if one thread touches an equivalence point, other threads can continue executing until touching the closest equivalence point under the control of the per-thread `ptrace` monitor (③ in Fig. 3).

There might be a case of one thread holding a lock reaching a breakpoint causing other threads never to reach an equivalence point. In such cases (*e.g.,* `pthread_mutex_lock()`), we temporarily disable the check code logic in the checker function on entering the critical region and re-enable the check code logic after leaving the critical region. As a result, whenever one thread holds a lock, other threads will not be allowed to pause at equivalent points. Other thread synchronization mechanisms, such as `pthread_join()`, may cause the *main thread* to be suspended at a location other than the semantic equivalence point. In such cases, once all children threads are paused, the runtime monitor rollbacks *main* thread execution context to an equivalence point right before the thread synchronization primitive (with `setjmp()`). Through this way, DAPPER ensures all threads are paused at semantic equivalence points.

Once the monitor seizes all threads, the DAPPER runtime sends a `SIGSTOP` to the process to completely pause its state. Next, the DAPPER runtime invokes CRIU [15] to dump the process into a process image, and then starts rewriting the image. To reduce the time for storing/restoring a process image to/from the hard disk, we checkpoint the process into an in-memory filesystem, *i.e.,* `tmpfs` [45].

### C. Process Rewriting

DAPPER's process rewriter leverages the CRIU checkpoint and restore mechanism to suspend the process for a short period of time and rewrites the CRIU-dumped process images to diversify the program state. By rewriting the (static) process image, we avoid the complications of dealing with potential race conditions that manifest in existing dynamic re-randomization systems [7], [10], [59].

The process rewriter supports transforming several process-level states, including updating memory page contents, changing the architecture-specific register descriptions and values, and rewriting the call stacks. For example, to *shuffle the stack slot layout*, the process rewriter locates the call stack memory, unwinds the stack, and rewrites stack frames. DAPPER also allows end-users to *transform architecture-related process states*, such as the register set and code pages. Since code pages can be loaded from the program binary, CRIU does not dump all code pages but only the execution context (*i.e.,* one or two code pages pointed by the program counter) into image files [15]. Therefore, the process rewriter replaces the current execution context with the corresponding code pages and updates the memory-mapped executable file in the ELF format for the target architecture. As a result, the new process will load the target architecture's code pages when the process is restored on that architecture.

Similarly, the process rewriter also transforms the register set. Since a target process is only paused at equivalence points, the process rewriter reads the *stack map* records at the checkpoint and translates the concrete register values from the source architecture into those of the target architecture. Fig. 4 shows a simple example of the stack map and how to translate live values using the stack map record. The function `add(int a, int b)` has two variables that are live on entering the function. The stack map records show that variables `a` and `b` are in register 3 and 14 on the `x86-64` platform and are in register 20 and 19 on the `aarch64` platform. Therefore, the DAPPER runtime pauses the process on entering function `add(int a, int b)` and rewrites live values in registers to translate the process state. Here the register numbers are encoded under the DWARF specification [12]. Note that the physical register numbers can differ between a RISC architecture CPU and a CISC CPU.[3] Moreover, a live value stored in a register on one architecture may be emitted into the call stack on another architecture. To solve this problem, we leverage the stack unwinding and rewriting technique to update the correct live values if they are on the stack.

*Thread Local Storage (TLS):* The symbol mapping for TLS structures is kept identical by our compiler for each target ISA. However, the offset from the start of the TLS structure referenced by the architecture-specific TLS register (the `FS` base register in `x86-64` and the `TPIDR` register in `aarch64`) is differently implemented within `libc` for each architecture. In DAPPER's design, we simply update the offset values while transforming the state to handle this TLS issue.

DAPPER's process rewriting technique differs from that of dynamic binary instrumentation (DBI) [9], [30] and dynamic binary transformation (DBT) [51] techniques. Both DBI and DBT techniques use a code cache to store the transformed code instructions and later execute them on the *local host*

[3]The RISC architecture simplifies hardware design by composing several simple instructions to implement a complex instruction of a CISC architecture. Thus, RISC architectures often tend to have more general-purpose registers.

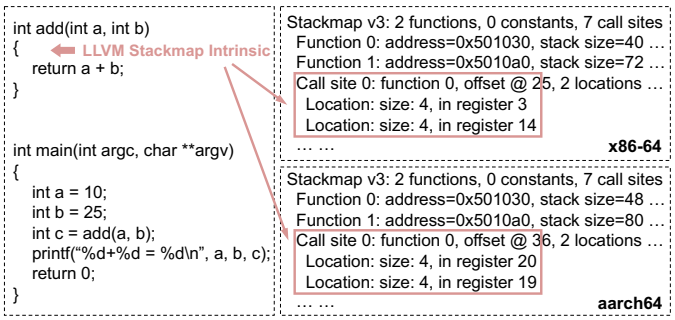


Fig. 4. Live value records and translation for cross-architecture rewriting using the stack map.

*machine*. These techniques are also designed to translate code of a different architecture on a single machine. In contrast, DAPPER rewrites the process by reloading code pages and rewriting process states. Thus, it can transform a whole process across machines of different architectures.

### D. Implementation

DAPPER's implementation includes a modified CRIU (version 3.15) [15] and an extended LLVM compiler toolchain (version 9.0) [27]. We implemented DAPPER's runtime monitor by extending the Linux `ptrace` interface. The runtime monitor invokes the CRIU/CRIT command line APIs to pause, transform, and restore the process in a multi-ISA, multi-node cluster environment.

*1) Compiler Extension:* We use LLVM/Clang to lower the application source code into LLVM IR code. We then implement an LLVM middle-end pass that analyzes the IR code and inserts equivalence points (*i.e.,* at all function boundaries). This pass also inserts intrinsics (*i.e.,* `llvm.experimental.stackmap`) in the LLVM IR code to generate stack map records at all the equivalence points [29]. We also modify LLVM/Clang to generate two ELF binaries, one for the `x86-64` architecture and one for the `aarch64` architecture, from the same application source code. This compilation process ensures that the machine code for different architectures is derived from the same LLVM IR code. Similar to existing works on process migration in heterogeneous-ISA systems [4], [5], [48], we also modify the GNU gold linker to *align* the addresses of all symbols (*i.e.,* functions, global data, static data) across the two binaries by padding `nop` instructions at the end of each symbol. This essentially creates a unified global virtual address space across all architectures, ensuring that all symbols have the same address across all architectures. Thus pointers (to symbols) are valid after migrating a process to a different architecture.

Our middle-end pass also inserts a `dapper_checker()` function at each equivalence point for pausing the process (threads) into a transformable state. This function reads a global flag and checks whether an end-user has initiated the process transformation. Once the flag is set, it raises the `SIGTRAP` signal and thereby pauses the thread. Note that an attacker cannot maliciously leverage a `SIGTRAP` signal for launching an attack because the runtime monitor also checks the program state against the stack map and ensures that the program is paused only at an equivalence point.

*2) Runtime for Process Rewriting:*

*a) `ptrace`-based monitor:* As previously discussed in Section III-B, DAPPER's runtime handles the `SIGTRAP` exception by attaching `ptrace`-based monitors to each thread and pauses all threads at equivalence points where they can be transformed. `ptrace` is a system call implemented in Unix-like systems which allows a process (hereby referred to as the *tracer*) to control the execution of another process (hereby referred to as the *tracee*) [33]. The tracer process can also read and manipulate the tracee process's memory and registers.

The monitor first gathers the thread ID information from the `/proc` file system on receiving the process rewriting request. It then iterates through the symbol table of the target processes and reads the stack map records. Next, the monitor attaches itself to the target process using the `PTRACE_ATTACH` command and changes the value of the transformation flag using the `PTRACE_POKEDATA` command. It then spawns one thread for each tracee thread. Each monitor thread now waits for its respective tracee thread to raise a `SIGTRAP` signal. Once all threads are paused, the monitor thread uses `PTRACE_DETACH` to detach from the tracee threads. The monitor's main thread now sends a `SIGSTOP` signal to the process to be transformed. The process is now ready to be dumped with CRIU.

*b) CRIU modification:* The runtime monitor is also responsible for calling our extended CRIU API to checkpoint and transform the process. CRIU checkpoints the target process and dumps it into several image files, primarily in the protocol buffer (`protobuf`) format [23]. Recall that CRIU's CRIT tool provides an API to decode, encode and print the CRIU process image. In DAPPER's implementation, we extensively extend this interface for rewriting a live process. Specifically, our extended CRIT API rewrites the following CRIU images:

*`core` and `files` image files.* The `core.img` file contains process information including thread information (register descriptions and values) and task states (signals and thread-local storage), among others. The `files.img` file contains opened files by the target process. For example, the binary name and location are saved in `files.img`. When transforming a process for executing on a different architecture, we first need to update the process information and the executable location to the corresponding architecture. DAPPER's process rewriter modifies these files to translate all architecture-related registers and update the ELF binary location.

*`pages` and `pagemap` image files.* The `pages.img` file contains raw page contents of the process, while the `pagemap.img` file contains information about which virtual memory regions are populated with data. Specifically, the `pages.img` file contains all populated data pages and one or two code page(s). These code pages contain the execution context pointed by the program counter. CRIU discards other code pages when checkpointing the process since those code

pages can be directly loaded from the binary when handling page faults. The `pagemap.img` file contains a list of entries describing the mapped memory information. Each entry describes the address of each VMA, the number of pages populated, and the VMA flags. The `pagemap.img` file can be used as a dictionary to index the raw pages in `pages.img`.

To transform processes across architecture boundaries, DAPPER has to change the architecture-specific states of the process image into a form that can be restored on the target architecture. CRIU dumps the register values of each thread in a separate file (`core-<thread id>.img`) and dumps raw memory bytes in one consolidated file (`pages-<N>.img`). The DAPPER's stack and register transformation API is a set of file reads and writes which set the live values within the memory dump. The API uses stack map metadata generated by the compiler for transformation. We implement the state transformation logic as a *crit* sub-command.

The transformation of register values is straightforward as the stack map metadata generated for each target ISA serves as a one-to-one mapping and the respective values can be copied from the source image files to the destination image files. To transform the stack, DAPPER first locates the stack by reading the stack pointer value (`sp` register in `core.img`) and the stack virtual memory (retrieved from the `pagemap.img` file). DAPPER unwinds the outermost stack frame inwards to update each frame. It reads the stack map metadata and copies the stack values from the source to the destination process image. While unwinding the frames, it also inserts the pointer into the caller's frame and the function's return address into the stack frame. DAPPER follows the destination architecture's ABI and retains the register-save procedure. If it encounters a pointer to the source stack, it implements a logic to map each live stack pointer to its respective stack allocation. Specifically, DAPPER first copies the stack allocation to its designated position in the destination stack and then generates a pointer to the stack allocation within the stack. Note that the virtual memory layout for the code and data section is already aligned for both ISAs (Section III-D1). Therefore, references other than the ones to the stack are directly copied from source to destination. Finally, DAPPER replaces the code page(s) with the corresponding code page(s) of the destination architecture.

Other transformation logics can be similarly implemented. For example, we use DAPPER to implement a stack shuffling system to defeat stack-based attacks. We leverage the static binary instrumentation (SBI) on the checkpointed process image for stack shuffling. In addition, we leverage existing open-source projects (*e.g.,* capstone [20]) for code disassembly and re-assembly. DAPPER allows us to easily identify the function call stack frames, permute the candidate stack objects, and update the code pages and stack map records to reflect the new permutation stack allocation offset.

*3) Optimize the Cross-Node Process Restoration:* After modifying all process images, DAPPER's runtime restores the process on the target machine. DAPPER supports two ways to restore the transformed process image: the *vanilla process restoration* and *post-copy memory restoration*. By default, CRIU requires an end-user to copy all process images to the target machine and restores the (transformed) process image into a running process. However, this can incur a significant service-interruption latency for applications consuming *a large memory footprint*. To solve this issue, CRIU has an option for post-copy memory migration (or lazy-migration) [14]. Specifically, it does not dump all memory pages but keeps most of them in memory at the source node; only a minimal set of task states that starts the process is copied to the destination node. When the process (restored on the destination node) starts accessing missing pages, CRIU handles the page faults through a page server, which retrieves the required page from the source node and serves it to the target process.

DAPPER also supports cross-architecture post-copy memory restoration. Specifically, we add an option to the CRIU lazy-migration handler and additionally dump the stack pages at the source node. The stack pages and other process image files mentioned earlier are enough for cross-architecture process transformation. We also observed significant latency improvement when migrating tasks with large dynamically allocated heap memory (in Section IV-A).

## IV. EVALUATION

We experimentally evaluated DAPPER to understand its performance, security use cases, and energy efficiency benefits. To demonstrate the usability of DAPPER in a cloud environment, we conducted our experiments on CloudLab [18] using `m510` and `m400` instances. We also measured the power consumption and energy efficiency when applying DAPPER to a small cluster of heterogeneous machines. We used an *x86-64* server with an Intel Xeon CPU E5-2620 v4 @ 2.10GHz (eight cores) and 32 GB RAM, and three Raspberry Pi boards. The Raspberry Pi is equipped with four cores of Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz and 2 GB RAM. Our evaluation benchmarks included the NAS Parallel Benchmark (NPB) suite's serial version (classes A and B), the Linpack benchmark, the Dhrystone benchmark, applications from PARSEC benchmark suites that are written in C, the Redis key-value store (v5.4.0), the Nginx web server (v1.3.9) and a K-means clustering application. All benchmarks were compiled using our modified compiler described in Section III-D1.

### A. Architecture-Level Program State Rewriting

We first evaluated DAPPER's performance by measuring the end-to-end time cost for process rewriting and cross-node process restoration. As shown in Fig. 5, the whole process transformation overhead includes the time to *pause the process* (checkpoint), *rewrite* the process image (recode), *copy* the transformed process image over the network (scp), and *resume* the paused process on the target architecture (restore).

We notice that the checkpoint and restoration time is relatively fast (less than 30 *ms*). There are two major time costs in current DAPPER implementation: rewrite process images and cross-machine copy of the transformed process images. For example, DAPPER takes an average of *253.69 ms* to rewrite (recode) the checkpointed benchmark processes on the

*x86-64* machine, and *1004.91 ms* on the *aarch64* machine. This is simply because the *x86-64* CPU has stronger micro-architectural properties (*e.g.,* higher clock speed); the transformation logic is identical. A point to be noted here is that process transformation to *x86-64* or *aarch64* architecture can be carried out on either platforms, *i.e.,* the target architecture is decided based on the architecture of the executable and not on the platform where DAPPER is being invoked. Therefore, we can always transform the process image on the most powerful machine (*i.e., x86-64*). The other significant cost comes from copying process images between different machines. Using the InfiniBand, it takes about 300 *ms* to copy process images.

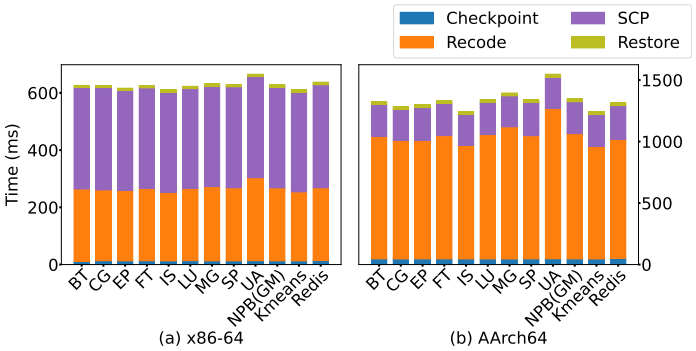


Fig. 5. Breakdown of DAPPER's time cost for cross-architecture process transformation.

We further measured the end-to-end execution time for DAPPER to relocate multi-threading PARSEC C applications across machine nodes of different architectures. We compared the total execution time of using DAPPER against native execution on each node (Fig. 6). Code execution on the *aarch64* machine takes longer than on the *x86-64* machine. This is simply because of the CPU frequency and the processing speed difference between *x86-64* and *aarch64* processors. With DAPPER, the total execution time lies in between the time of native execution on each node.

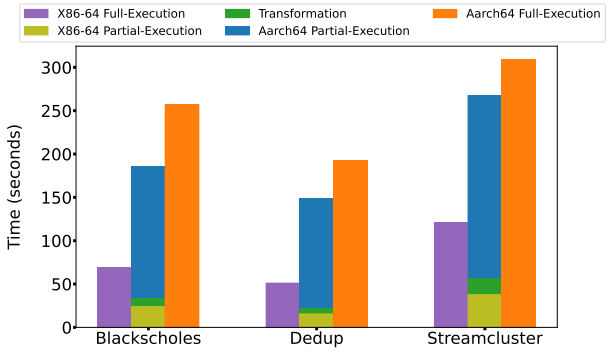


Fig. 6. Total execution time of DAPPER on multi-threading PARSEC C applications compared with native execution.

*a) Lazy-migration.:* For applications with larger memory footprint, DAPPER also provides a post-copy memory migration mode. It copies a small set of process images and then starts the program. Therefore, the time used for process checkpointing and image copying can be *significantly reduced* (`checkpoint` and `scp` in Fig. 7). The time used to rewrite the process in lazy-migration mode is *slightly better* because it takes less time to search and update the stack memory in the process image (`recode` in Fig. 7).

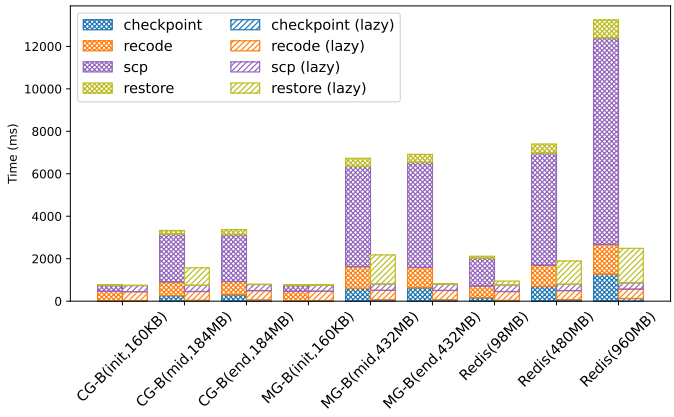


Fig. 7. Performance comparison of DAPPER vanilla migration v.s. lazy-migration (*x86-64* →*arm64*, with InfiniBand).

The restoration time measurement is more complicated. The lazy-migration process restoration immediately restores the process by loading the minimal execution context (takes about 8 *ms*). It then starts retrieving pages on demand from the original node. The time to retrieve pages from the original node is not easy to measure since that time is concealed in the post-migration execution. We still managed to estimate this indirect restoration cost. Specifically, since CG and MG applications terminate after a while, we can use the total execution time and subtract the measurable time of `checkpoint`, `scp` and `recode` to approximate the indirect restoration cost. The Redis server executes in an infinite loop. Therefore, we cannot subtract the time cost. Instead, we read the timestamp from the page server's log to estimate the indirect restoration cost.

For HPC applications such as CG and MG, we checkpointed at the beginning, in the middle, and towards the end of the execution, respectively (`init`, `mid` and `end` in Fig. 7). The lazy migration performs similarly to the vanilla checkpoint and restore mechanism when triggering the migration at the beginning of the execution. However, lazy migration can perform better after the program has fully launched. This is possibly due to the more enormous (heap) memory usage after the program warms up. The overall time is reduced when applications execute toward the end as it requires fewer memory pages to transform to finish the program execution. We observed similar performance patterns transforming a Redis server with different in-memory database sizes. In general, lazy migration significantly reduces the restoration latency and saves the total execution time.

*b) Energy efficiency and throughput improvement on a heterogeneous cluster.:* We evaluated energy efficiency of scheduling jobs on heterogeneous cores by measuring the number of executed jobs per power unit (kJ). Specifically,

we chose four benchmarks in NPB benchmark suites (class-B) and created an infinite queue of jobs to simulate a batch-processing HPC scenario [40]. We experimented on an Intel Xeon server and three Raspberry Pi boards and measured the energy consumption of processing the job queue for 30 minutes. The power consumption was measured using a SURAIELEC Energy Watt Meter. We also implemented a simple scheduler to evict tasks to one Raspberry Pi or three Raspberry Pis when the *x86-64* server runs out of CPU resources (more running jobs than CPU cores). Fig. 8 shows this result. When dynamically evicting jobs to low-end embedded boards, DAPPER improves the energy efficiency for 15% - 39%, depending on the workload types. We observed that the Raspberry Pi running three job threads consumes 5.1W power, whereas the Xeon server consumes 108W to run seven threads. The throughput improvement is similar to energy efficiency: DAPPER improves the throughput for 37% - 52% on this hybrid-architecture environment. Both results proved that DAPPER could effectively utilize heterogeneous compute resources by dynamically and seamlessly relocating job threads between machine nodes of different hardware architectures.

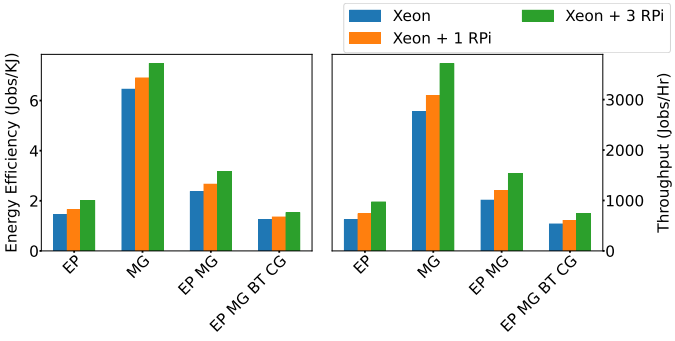


Fig. 8. Energy efficiency and throughput improvement of dynamically migrating (evicting) processes to Raspberry Pis.

### B. Stack State Shuffling

We also implemented a process image rewriting tool using DAPPER to permute stack allocations in the checkpointed process image and transformed source binary. Fig. 9 shows the average time taken for stack shuffling and the breakdown of the time cost, for all the benchmarks, on both *x86-64* and *aarch64* architectures. The time taken by the shuffle stage is proportional to the size of the code section in both the checkpointed process and the transformed source binary. DAPPER took an average of *573 ms* on *x86-64* and *3.2 s* on *aarch64* to shuffle the stack allocations and update the corresponding code pages and stack map records.

We quantify the benefits of DAPPER framework's stack shuffling by measuring the permutation entropy. Fig. 10 quantifies the degree of randomness introduced by DAPPER in terms of *bits of entropy* for all the benchmarks on the *x86-64* and *aarch64* servers. Bits of entropy represents the number of pairwise stack allocation shuffles in a stack frame. DAPPER

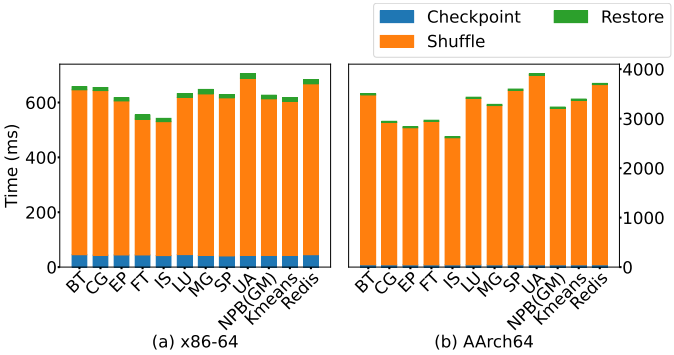


Fig. 9. Breakdown of DAPPER's time cost for stack shuffling process transformation.

permutes stack allocations by pairing them based on their allocation size and then permuting every such pair. Shuffling a stack frame with $n$ bits of entropy results in $1+(2n-1)!!$ [55] possible stack frames, providing an attacker with $1/(2 \times n)$ probability of guessing a correct stack allocation. For example, four bits of entropy represent shuffling eight stack allocations, resulting in $1+(2\times4-1)!!$ = *106* possible stack frames with $1/(2 \times 4)$ = *0.125* probability of guessing the location of a single stack allocation by the attacker. If a data-oriented attack (DOA) [22], [24] requires manipulating three stack allocations for building an expressive payload, the probability of such an attack succeeding is $0.125^3$ (*i.e.,* 0.19%).

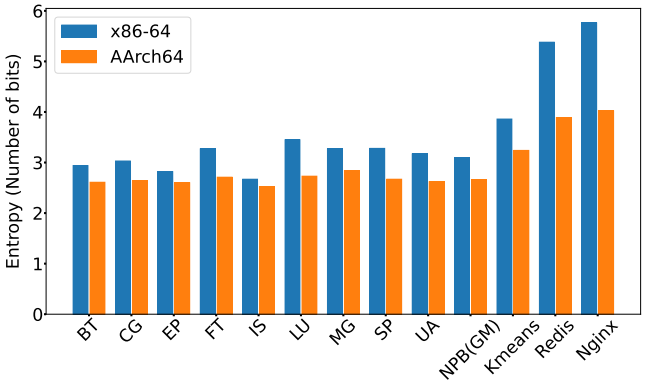


Fig. 10. Average bits of entropy introduced by DAPPER due to stack shuffling.

Fig. 10's y-axis represents the average bits of entropy across all stack frames in all functions. As the figure shows, DAPPER introduces 5.76 bits of entropy in Nginx, 5.38 bits in Redis, and 3.09 bits in the NPB benchmarks on the *x86-64* architecture, with an average of *4.74 bits* across all benchmarks. On the other hand, DAPPER introduces 4.02 bits of entropy in Nginx, 3.32 bits in Redis, and 2.65 bits in the NPB benchmarks for *aarch64*, with an average of *3.33 bits* across all benchmarks. DAPPER introduces lower entropy on *aarch64* than *x86-64* due to the exclusion of load and store pair instructions accessing the stack frames. Permuting stack offsets of these stack references require re-encoding these instruction pairs to

single stack reference instructions, which we scoped out in the current effort. DAPPER's future implementation can further increase the entropy by considering these instructions.

Next, we demonstrate the security benefits of DAPPER's runtime process rewriting technique by evaluating it against exploits showcased in DOP [22] and BOPC [24], and known stack-based CVEs [13], [42]. The Min-DOP exploit in [3] is a synthetic DOP attack that leverages memory vulnerability exposed through integer underflow and incorrect type checking to perform arbitrary memory reads. In addition, it exposes an out-of-bound stack memory vulnerability to perform arbitrary memory writes. Finally, the exploit leverages arbitrary memory read and write capabilities together to construct expressive payloads to perform privilege escalation and confidential data leak operations. DAPPER mitigates privilege escalation and confidential data leak exploits conducted using Min-DOP through disruption of DOP gadget chaining and dispatching. DAPPER's stack shuffling causes the relocation of exploit-sensitive data around the overflowed buffer, resulting in incorrect gadget chaining and dispatching. In addition, by transparently transforming the architecture state, DAPPER prevents the payload from succeeding since live values on the stack and registers are completely relocated.

The Block-Oriented Programming Compiler (BOPC) [24] is a tool that automatically synthesizes Turing-complete DOP payloads expressed by the attacker in a high-level language called the SPloit language (SPL). We utilize the BOPC tool on the Nginx web server to generate DOP payloads for exploits involving arbitrary memory reads and writes, register reads and writes, and execve shell spawns. Exploits based on memory or register read payloads exercise DOP gadget chains and dispatcher blocks to read stack references into registers. In contrast, exploits based on memory or register write payloads exercise DOP gadget chains and dispatcher blocks to update stack references from register values. DAPPER mitigates these exploits by relocating the stack allocations referred to by gadget chains or dispatcher blocks. Since DAPPER relocates data-flow-critical non-control data through stack shuffling, BOPC exploits are rendered ineffective. Exploits constructed to spawn a shell first identify basic blocks invoking the `execve()` system call and stitch gadgets through dispatcher blocks to perform memory reads and register writes. DAPPER mitigates these attacks by disrupting arbitrary memory read and register write gadgets.

We also evaluated DAPPER against real-world exploits exposed through *CVE-2015-4335* which affects the Redis key-value store (v5.4.0). Our experiment leveraged an existing exploit framework using redis-rce [43]. The exploit constructs *loadstring* ROP gadgets to load unsafe Lua bytecodes and escape Lua's enforced sandboxing. The exploit is initiated through arbitrary memory read and write capability and can result in exploits such as privilege escalation and confidential data leaks. DAPPER mitigates these exploits through stack shuffling which introduces enough entropy to break arbitrary read and write capabilities required to construct the ROP gadget chains. We also observed that DAPPER mitigated a synthetic arbitrary code execution exploit on the Nginx web server (v1.3.9) exposed through a stack buffer overflow vulnerability recorded in *CVE-2013-2028*.

### C. Comparison against Competitor Techniques

We also evaluated DAPPER against competitor techniques [1], [5], [47], [48], [59], [62] in terms of the attack surface reduction. The source code of many of them [1], [47], [48], [59] is not publicly available at the time of this paper's writing. Therefore, we measured DAPPER's attack surface reduction compared with two open-source works [5], [62]. The Popcorn Linux software stack [5] enables live process migration in a multi-ISA environment. H-Container [62] extends Popcorn Linux to a container environment and removes Popcorn Linux's specialized kernel from the trusted computing base (TCB). We measured the attack surface of *application program binaries* used in these systems by calculating the ROP gadget count. Fig. 11 shows the percentage reduction of ROP gadgets in the benchmark binaries generated through DAPPER for *x86-64* and *aarch64*, respectively. The baseline is the ROP gadget count in the binaries generated by Popcorn Linux [5]. DAPPER reduces the number of ROP gadgets by an average of *59.28%* for *x86-64* and *71.91%* for *aarch64*, respectively. This is mainly because DAPPER's lightweight framework rewrites a target process externally. Thus, the state transformation logic is not included in the program's address space and is not exposed to an external attacker.

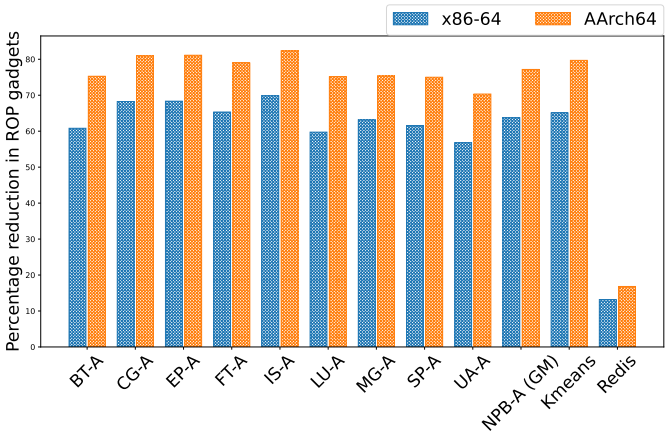


Fig. 11. DAPPER's attack surface reduction in terms of ROP gadget count compared to that of Popcorn Linux [5] (and H-Container [62]).

## V. RELATED WORK

The first category of related work includes various *techniques for (dynamic) software transformation* [26]. Software transformation techniques create program variants that function the same with different software structures (*i.e.,* code and/or data layouts). There are many research efforts to dynamically diversify a program, such as address space layout randomization (ASLR) [19], runtime code randomization [7], [59], dynamic software customization and debloating [32], and data space randomization [1], [8], [31], among others.

An important design problem is ensuring that the transformed program functions like the original one. To solve this problem, existing approaches generate auxiliary code or metadata to facilitate program translation. During runtime, an (inline) code monitor periodically randomizes the code layout and updates the auxiliary code and metadata [7], [59]. However, the inline code monitor can be a potential attack target and often needs additional security protection [59].

The industry and architecture community have recently explored utilizing architecture-level diversity to improve energy utilization, security and reduce data-center costs [5], [38], [40], [41], [47], [48], [63]. For example, architecture researchers have shown that single chip multi-ISA CPUs can increase entropy with inter-ISA program state randomization [47], [52]. The multi-architecture platforms can also improve power utilization by scheduling workload based on architecture-beneficial code regions [5], [40], [48]. Efforts such as HSA [44], Venkat *et al.* [48], and Popcorn Linux [5] demonstrated process migration across ISA-different CPUs by using a common data format and limiting the amount of ISA/ABI-specific program state that needs to be converted at runtime, thereby reducing migration overheads. By using a common address space layout across different ISAs, these systems align the memory layout (*e.g.,* stack, heap, TLS), function addresses, and data addresses in the address space so that the validity of pointers are preserved across ISAs.

Since machine code must be different for different ISAs, Popcorn Linux compiles the same program into ISA-specific binaries and ensures the same starting address for functions in the *.text* section of all output binaries [5]. It also assumes that memory is shared across CPUs of different ISAs and patches the Linux kernel to map the process address space on shared memory so that multiple CPUs of different ISAs can host threads belonging to the same process, similar to CPUs of the same ISA hosting threads in a multi-core chip. In addition, cross-ISA transformation logic is injected into the address space of each process. This logic leverages special systems calls in the Linux kernel to place values in appropriate registers before scheduling each thread. This results in a relatively large attack surface, including the inline code transformer and extended kernel page-sharing mechanism, which makes Popcorn Linux unsuitable for practical use [5]. In contrast, DAPPER obliterates the code transformer from the target process, significantly reducing the attack surface and enabling additional security hardening, such as stack shuffling. Moreover, DAPPER is lightweight and has minimal modification on the system software stack, making it easier to use.

The second category of related work includes (dynamic) binary translation [17], [53], binary lifting [2], [39], and re-compilation [2], [60]. Binary translation aims to convert the code of an executable file into enhanced code with additional functionalities or new code that can run on another architecture [53]. There exists different techniques for implementing binary translation. *Static binary rewriting* inserts instruction snippets into Commercial-Off-The-Shelf (COTS) binaries for software security patching, binary repairing, and hardening [17], [60]. Binary lifting and re-compilation leverage the metadata present in current stripped *x86-64* and *aarch64* binaries to generate a complete assembly code and further lifts the assembly into layout-agnostic intermediate representation for code transformation [2], [39], [60]. For example, McSema is an executable lifter that translates (*i.e.,* "lifts") executable binaries from native machine code to LLVM bitcode [39]. After obtaining the intermediate representation of the binary, users can apply compiler-based optimization passes on top of the recovered representation [2]. DAPPER shares similarities with binary lifting and re-compilation. However, instead of transforming the binary as in binary lifting and re-compilation, DAPPER pauses and rewrites the process at runtime. Thus, DAPPER can be viewed as a mechanism for *dynamic* software transformation.

## VI. CONCLUSIONS

We presented DAPPER, a lightweight and extensible framework that allows dynamic transformation of the program and architectural state. DAPPER utilizes the process checkpoint/restore mechanism to rewrite the process state on the snapshotted checkpoint. Compared with existing approaches, DAPPER is lightweight and can support more generic scenarios. We have built a prototype of DAPPER to transform the program state and architectural states across *x86-64* and *aarch64* architectures. Our evaluation shows that DAPPER's cross-architecture process rewriting capability can improve the energy efficiency and throughput for batch processing HPC workloads by up to 39% and 52% when dynamically evicting jobs to low-power Raspberry Pi boards. It adds about 4 bits of program state entropy when rewriting the target process's stack slots, defeating several classes of CVEs. Moreover, it reduces the attack surface by as much as ≈60-72% over prior arts due to the external process rewriting mechanism.

## ACKNOWLEDGMENT

We thank the anonymous reviewers for their insightful comments, which have greatly improved the paper. This work is partly supported by the US Office of Naval Research under grants N00014-18-1-2022, N00014-19-1-2493, and N00014-22-1-2672, and by the US National Science Foundation (NSF) under grant CNS 2127491. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.

## REFERENCES

[1] Misiker Tadesse Aga and Todd M. Austin. Smokestack: Thwarting DOP Attacks with Runtime Stack Layout Randomization. In Mahmut Taylan Kandemir, Alexandra Jimborean, and Tipp Moseley, editors, *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, pages 26–36. IEEE, 2019.

[2] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, Herbert Bos, and Michael Franz. BinRec: dynamic binary lifting and recompilation. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 36:1–36:16. ACM, 2020.

[3] Miguel A. Arroyo. Minimal Data-Oriented Programming (DOP) Vulnerable Server + Exploits. https://github.com/mayanez/min-dop, 2020.

[4] Antonio Barbalace, Mohamed L. Karaoui, Wei Wang, Tong Xing, Pierre Olivier, and Binoy Ravindran. Edge Computing: The Case for Heterogeneous-ISA Container Migration. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 73–87, New York, NY, USA, 2020. Association for Computing Machinery.

[5] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. Breaking the boundaries in heterogeneous-ISA datacenters. In *ACM SIGPLAN Notices*, volume 52, pages 645–659. ACM, 2017.

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008.

[7] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 268–279. ACM, 2015.

[8] Christopher Blackburn, Xiaoguang Wang, and Binoy Ravindran. Rave: A modular and extensible framework for program state rerandomization. In Hamed Okhravi and Cliff Wang, editors, *Proceedings of the 9th ACM Workshop on Moving Target Defense, MTD 2022, Los Angeles, CA, USA, 7 November 2022*, pages 3–10. ACM, 2022.

[9] Derek Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, Sept 2004.

[10] Yue Chen, Zhi Wang, David Whalley, and Long Lu. Remix: On-demand live randomization. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 50–61. ACM, 2016.

[11] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286, 2005.

[12] Eager Consulting. The DWARF Debugging Standard. https://dwarfstd.org/, 2021.

[13] MITRE Corporation. CVE-2015-4335. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-4335, 2015.

[14] CRIU. Lazy migration. https://criu.org/Lazy_migration, 2022.

[15] CRIU. Checkpoint Restore in Userspace. https://criu.org/Main_Page, 2023.

[16] CRIU. CRIT: CRiu Image Tool. https://criu.org/CRIT, 2023.

[17] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 151–163. ACM, 2020.

[18] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[19] Jake Edge. Linux Kernel Address Space Layout Randomization. http://lwn.net/Articles/569635/, 2013.

[20] Capstone Engine. Capstone: The Ultimate Assembler. https://www.capstone-engine.org, 2022.

[21] Michael Hicks and Scott Nettles. Dynamic software updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1049–1096, 2005.

[22] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986. IEEE, 2016.

[23] Google Inc. Protocol Buffers. https://developers.google.com/protocol-buffers, 2022.

[24] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block Oriented Programming: Automating Data-Only Attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1868–1882, New York, NY, USA, 2018. Association for Computing Machinery.

[25] Yufei Jiang, Can Zhang, Dinghao Wu, and Peng Liu. Feature-based software customization: Preliminary analysis, formalization, and methods. In *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*, pages 122–131. IEEE, 2016.

[26] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated Software Diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, 2014.

[27] The LLVM Compiler Infrastructure. http://llvm.org/.

[28] LLVM. Extending LLVM: Adding instructions, intrinsics, types, etc. https://llvm.org/docs/ExtendingLLVM.html#intrinsic-function, 2022.

[29] LLVM. Stack maps and patch points in LLVM. https://llvm.org/docs/StackMaps.html, 2022.

[30] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 190–200. ACM, 2005.

[31] Robert Lyerly, Xiaoguang Wang, and Binoy Ravindran. Dynamic and Secure Memory Transformation in Userspace. In *Computer Security - ESORICS 2020 - 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14-18, 2020, Proceedings, Part I*, volume 12308 of *Lecture Notes in Computer Science*, pages 237–256. Springer, 2020.

[32] Abhijit Mahurkar, Xiaoguang Wang, Hang Zhang, and Binoy Ravindran. Dynacut: A framework for dynamic and adaptive program customization. In *Proceedings of the 24th International Middleware Conference, Middleware 2023, Bologna, Italy, December 11-15, 2023*, pages 275–287. ACM, 2023.

[33] Michael Kerrisk. ptrace(2) — Linux manual page. https://man7.org/linux/man-pages/man2/ptrace.2.html, 2021.

[34] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for c. *ACM SIGPLAN Notices*, 41(6):72–83, 2006.

[35] NGD. Newport computational storage platform. https://ngdsystems.com/, 2019. Last accessed April 2022.

[36] NVIDIA. ConnectX-5: Advanced Offload Capabilities for the Most Demanding Applications. https://www.nvidia.com/en-us/networking/ethernet/connectx-5/. Last accessed April 2022.

[37] NVIDIA. Innova-2 Flex: Advanced Programmability Delivers Acceleration and Performance. https://www.nvidia.com/en-us/networking/ethernet/innova-2-flex/. Last accessed April 2022.

[38] NVIDIA. NVIDIA BlueField Data Processing Units. https://www.nvidia.com/en-us/networking/products/data-processing-unit/. Last accessed April 2022.

[39] Trail of Bits. McSema. https://github.com/lifting-bits/mcsema, 2022.

[40] Pierre Olivier, A. K. M. Fazla Mehrab, Stefan Lankes, Mohamed Lamine Karaoui, Robert Lyerly, and Binoy Ravindran. HEXO: offloading HPC compute-intensive workloads on low-cost, low-power embedded systems. In Jon B. Weissman, Ali Raza Butt, and Evgenia Smirni, editors, *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2019, Phoenix, AZ, USA, June 22-29, 2019*, pages 85–96. ACM, 2019.

[41] Oracle. Oracle cloud infrastructure: Ampere A1 compute, 2021. https://www.oracle.com/cloud/compute/arm/, Last accessed April 2022.

[42] Inc. Red Hat. CVE-2013-2028. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2013-2028, 2013.

[43] Ridter. CVE-2015-4335 Redis exploit. https://github.com/Ridter/redis-rce, 2021.

[44] Phil Rogers and A Fellow. Heterogeneous system architecture overview. In *Hot Chips Symposium*, pages 1–41, 2013.

[45] Christoph Rohland. Tmpfs. https://www.kernel.org/doc/html/latest/filesystems/tmpfs.html, 2020.

[46] Samsung. SmartSSD: Computational Storage Drive. https://samsungsemiconductor-us.com/smartssd/. Last accessed April 2022.

[47] Ashish Venkat, Sriskanda Shamasunder, Hovav Shacham, and Dean M Tullsen. Hipstr: Heterogeneous-ISA program state relocation. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 727–741. ACM, 2016.

[48] Ashish Venkat and Dean M Tullsen. Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor. *ACM SIGARCH Computer Architecture News*, 42(3):121–132, 2014.

[49] David G. von Bank, Charles M. Shub, and Robert W. Sebesta. A unified model of pointwise equivalence of procedural computations. *ACM Trans. Program. Lang. Syst.*, 16(6):1842–1874, nov 1994.

[50] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L Scott. Proactive process-level live migration in hpc environments. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12. IEEE, 2008.

[51] Wenwen Wang, Stephen McCamant, Antonia Zhai, and Pen-Chung Yew. Enhancing Cross-ISA DBT Through Automatically Learned Translation Rules. In Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar, editors, *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, pages 84–97. ACM, 2018.

[52] Xiaoguang Wang, SengMing Yeoh, Robert Lyerly, Pierre Olivier, Sang-Hoon Kim, and Binoy Ravindran. A Framework for Software Diversification with ISA Heterogeneity. In Manuel Egele and Leyla Bilge, editors, *23rd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2020, San Sebastian, Spain, October 14-15, 2020*, pages 427–442. USENIX Association, 2020.

[53] Wikipedia. Binary translation. https://en.wikipedia.org/wiki/Binary_translation, 2022.

[54] Wikipedia. Debugger. https://en.wikipedia.org/wiki/Debugger, 2023.

[55] Wikipedia. Double factorial. https://en.wikipedia.org/wiki/Double_factorial, 2023.

[56] Wikipedia. LINPACK benchmarks. https://en.wikipedia.org/wiki/LINPACK_benchmarks, 2023.

[57] Wikipedia. Live migration. https://en.wikipedia.org/wiki/Live_migration, 2023.

[58] Wikipedia. Live-variable analysis. https://en.wikipedia.org/wiki/Live-variable_analysis, 2023.

[59] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *OSDI*, pages 367–382, 2016.

[60] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. Egalito: Layout-Agnostic Binary Recompilation. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 133–147. ACM, 2020.

[61] Parkson Wong and R Der Wijngaart. Nas parallel benchmarks i/o version 2.4. *NASA Ames Research Center, Moffet Field, CA, Tech. Rep. NAS-03-002*, page 91, 2003.

[62] Tong Xing, Antonio Barbalace, Pierre Olivier, Mohamed L. Karaoui, Wei Wang, and Binoy Ravindran. H-Container: Enabling Heterogeneous-ISA Container Migration in Edge Computing. *ACM Trans. Comput. Syst.*, mar 2022.

[63] ZDNET. AWS Graviton2: What it means for ARM in the data center, cloud, enterprise, AWS, 2019. https://www.zdnet.com/article/aws-graviton2-what-it-means-for-arm-in-the-data-center-cloud-enterprise-aws/, Last accessed April 2022.