Catamaran: Low-Overhead Memory Safety Enforcement via Parallel Acceleration

Yiyu Zhang[*]
Nanjing University, China
zhangyy0721@smail.nju.edu.cn

Tianyi Liu[*]
Nanjing University, China
tyi.liu@smail.nju.edu.cn

Zewen Sun[*]
Nanjing University, China
sunzew@smail.nju.edu.cn

Zhe Chen
Nanjing University of Aeronautics and Astronautics, China
zhechen@nuaa.edu.cn

Xuandong Li[*]
Nanjing University, China
lxd@nju.edu.cn

Zhiqiang Zuo[*][†]
Nanjing University, China
zqzuo@nju.edu.cn

ABSTRACT

Memory safety issues are the intrinsic diseases of C/C++ programs. Dynamic memory safety enforcement as the dominant approach has an advantage in high effectiveness, yet suffers from prohibitively high runtime overhead. Existing attempts to reduce the overhead are either labor-intensive, tightly dependent on specific hardware/compiler support, or poorly effective.

In this paper, we propose a novel technique to reduce time overhead by executing the dynamic checking code in parallel. We leverage static dependence analysis and dynamic profit analysis to identify and dispatch the potential code to separate threads running simultaneously. We implemented a tool called Catamaran and evaluated it over a rich set of benchmarks. The experimental results validate that Catamaran is able to significantly reduce the runtime overhead of the existing dynamic tools, without sacrificing capability of memory safety enforcement.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**; Software testing and debugging; • **Theory of computation** → *Program analysis*.

KEYWORDS

memory safety, program analysis, parallel acceleration, compiler optimization

ACM Reference Format:
Yiyu Zhang, Tianyi Liu, Zewen Sun, Zhe Chen, Xuandong Li, and Zhiqiang Zuo. 2023. Catamaran: Low-Overhead Memory Safety Enforcement via Parallel Acceleration. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598098

[*]Also with State Key Laboratory for Novel Software Technology at Nanjing University.
[†]Corresponding author.

1 INTRODUCTION

The C/C++ languages are widely used for implementing a rich spectrum of systems software thanks to their high performance and low-level control of system resources (*e.g.*, memory). However, due to the focus on performance and flexibility, memory safety is not guaranteed at the language level to ensure that programmers manipulate memory correctly and safely. As such, memory safety violation is prevalent in C/C++ programs, which always leads to not only various subtle bugs (*e.g.*, silent memory corruption, difficult-to-diagnose crashes, and invalid outputs), but also security vulnerabilities (*e.g.*, buffer overflow, use-after-free). The report from MSRC reveals that about 70% of the annual Microsoft patches are for memory security vulnerabilities since 2004 [28]. Google researchers also conducted a study showing that around 70% of the security vulnerabilities marked as "high" or "severe" in the Chromium project are memory safety problems [17].

To ensure memory safety, one dominant and effective class of approaches [3, 9, 31, 40] is to dynamically check the validity of memory accesses at runtime. For this purpose, the additional code snippets (a.k.a., meta-operations) are embedded into the original program so as to record, update and check against the meta-data (*e.g.*, boundary information) of each memory region allocated. For ease of presentation, we use the term "dynamic checking code" in the rest of the paper to represent all these additional code snippets inserted. On the one hand, these approaches are effective since all the memory accesses are dynamically checked against the precisely tracked meta-data of memory regions. On the other hand, the intensive dynamic checking code results in prohibitively high time overhead, which severely undermines their practicability.

State of the Art. To lower the high time overhead, numerous studies have been presented to reduce or accelerate the execution of dynamic checking code inserted. The existing work can be roughly classified into three categories. The first category reduces redundant checking code by automatically analyzing or manually annotating the source code [13, 47, 48]. Unfortunately, they are limited by either the imprecision and incompleteness of program analyses, or the intensive manual effort of annotations. The second is the hardware-based approach which manipulates specific hardware to accelerate the execution of dynamic checking code [30, 42, 45]. However, these approaches can hardly be general since 1) they rely on the specific compiler to support the extended ISA; and 2) they cannot support all types of memory safety errors. The third category is to decouple the execution of dynamic analysis from the original execution, and offload it on other cores/processes [8, 24, 33, 36]. However, these approaches either require manual intervention to specify the code decoupled a priori [24], or need frequent expensive replay [33], or suffer from heavy communication cost due to intensive data/log passing [8, 43].

Insight & Our Approach. Different from all the existing work, we propose a novel approach to reduce the runtime overhead by executing dynamic checking code in parallel.

We observed that not all the code snippets in the original program and the dynamic checking code are dependent on each other. In fact, it is very likely that there exists no dependence at all between dynamic checking code and original code or among dynamic checking code themselves. Therefore, there is a large potential benefit for making certain code snippets of the original code and dynamic checking code run in parallel. Additionally, as the multi-core CPUs are prevalent in commodity PC, CPU cores can be rarely 100% utilized over a long period of time [6, 26]. In the general computing scenario, we have reasons to believe that the host machine would have extra CPU resources to be exploited.

Based on the above insights, we propose to accelerate the dynamic checking code via fine-grained thread-level parallelism, thus lowering the time overhead. In brief, our approach treats each small fragment of dynamic checking code as a potential parallel task, automatically identifies all the potential tasks to be parallelized, and produces an optimized schedule with correctness guarantee by leveraging rigorous and sound static analysis (*i.e.*, def-use analysis, happens-before analysis, and alias analysis). Moreover, it adopts thread-level parallelism to enable in-memory information sharing among parallel tasks, thus avoiding heavy communication cost.

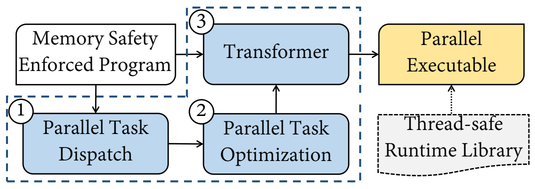

Memory Safety Enforced Program
③
Transformer
Parallel Executable
①
Parallel Task Dispatch
②
Parallel Task Optimization
Thread-safe Runtime Library

Figure 1: Workflow of Catamaran.

We implement our approach as a tool called Catamaran, whose workflow is illustrated by Figure 1. Given a memory safety enforced C/C++ program where the intensive dynamic checking code is inserted, Catamaran firstly conducts the parallel task dispatch (cf. ①). We leverage a rigorous def/use analysis to achieve the dependencies between each piece of dynamic checking code and the original code. Based on the dependencies, Catamaran dispatches each piece of dynamic checking code which can run in parallel as a parallel task executed by a separate thread. However, the above naive task dispatching could instantiate too many tiny tasks, which severely undermines the overall performance gain. To this end, Catamaran exploits a profit analysis-based optimization (cf. ②). We introduce a cost model for profit analysis to measure the approximate cost of each parallel task. Based on this model, we seek the maximum profit by performing the appropriate parallel task management, *e.g.*, merging small tasks, rescheduling certain tasks without violating the dependence restriction. Finally, based on the optimized parallel tasks, Catamaran transforms the source code into multi-threaded executables leveraging the thread pool mechanism (cf. ③). We also implement a thread-safe version of meta-operation runtime library, which can safely access and manipulate meta-data in a parallel way. Each step of Catamaran will be discussed in details shortly.

Results & Contributions. We have implemented Catamaran on top of the LLVM compiler infrastructure [23]. We selected three representative memory safety enforcement tools, namely SoftBound-CETS [31], MoveC [9], and AddressSanitizer [40] as the candidates to be accelerated, and conducted the experiments over a set of benchmarks including SPEC CPU 2006, SPEC CPU 2017, MoveC-MSBench and 8 CVEs. The experimental results show that Catamaran with 4 threads reduces runtime overhead by 46% to 224% on average with negligible memory overhead. Moreover, Catamaran does not diminish the capability of detecting memory errors compared to the baseline memory enforcement tool.

In summary, we make the following contributions.

- We propose a novel approach that accelerates the dynamic checking code via multi-threading parallelism.
- We implement a prototype of Catamaran that enables us to reduce the runtime overhead of the existing dynamic memory safety enforcement tools.
- We conduct the comprehensive evaluations which validate that Catamaran significantly reduces the time overhead of memory safety enforcement tools, without sacrificing enforcement capability.

Outline. The rest of the paper is organized as follows. §2 gives the necessary background of dynamic memory safety enforcement and overview of Catamaran. §3 formalizes the problem we have to solve and §4 describes the approach we proposed, followed by the implementation of Catamaran in §5. We present the empirical evaluations in §6. Certain issues are discussed in §7. We talk about the related work in §8. Finally, §9 concludes.

2 BACKGROUND & OVERVIEW

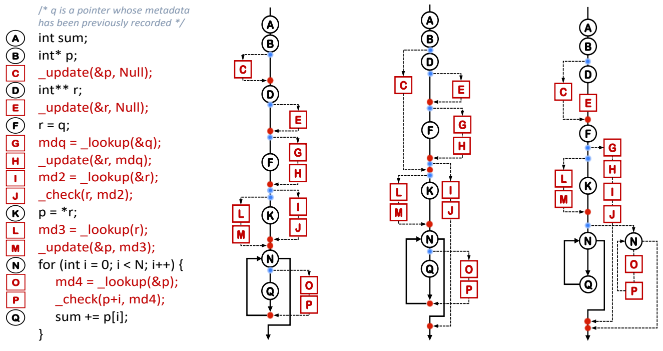
2.1 Background

Dynamic approaches enforce memory safety by dynamically checking the validity of memory accesses at runtime. They follow the same basic schema which inserts the pre-defined meta-operations into each memory access point of the subject program. Table 1 lists the three meta-operations used for recording, updating, and checking the meta-data of memory regions allocated.

Table 1: Three basic meta-operations used.

Operation	Input	Output	Description
_update	key, meta-data	void	update the key and its associated meta-data
_lookup	key	meta-data	find and return meta-data associated with key
_check	ptr, meta-data	void	validate the memory access against meta-data

Without loss of generality, we take the approaches preventing spatial and temporal memory errors (*e.g.*, buffer overflow, use-after-free) as an example to explain how they work. For each pointer *ptr* referring to a memory region allocated in a program, we maintain its respective boundary and temporal status meta-data which can be represented as a tuple $\langle start, end, status \rangle$. All the information of all memory regions involved can be stored in a lookup table where the key corresponds to the address of pointer (*i.e.*, *&ptr*), and the value is the respective meta-data. The *_update* function is responsible for storing/updating the meta-data associated with a particular key in the table. The *_lookup* function takes the key as input and returns the respective meta-data associated with the key. The *_check* function takes the meta-data and the current value of the pointer as input, and checks if the pointer goes beyond the valid boundary of the memory region to which the pointer refers and whether the pointer's status is legal.



```
    /* q is a pointer whose metadata
    has been previously recorded */
A   int sum;
B   int* p;
C   _update(&p, Null);
D   int** r;
E   _update(&r, Null);
F   r = q;
G   mdq = _lookup(&q);
H   _update(&r, mdq);
I   md2 = _lookup(&r);
J   _check(r, md2);
K   p = *r;
L   md3 = _lookup(r);
M   _update(&p, md3);
N   for (int i = 0; i < N; i++) {
O       md4 = _lookup(&p);
P       _check(p+i, md4);
Q       sum += p[i];
    }
```

(a) enhanced code **(b) intra** **(c) inter** **(d) optimized**

Figure 2: An example showing how Catamaran works; the solid line indicates the main thread, whereas dotted lines indicate parallel threads forked; the blue and red dots indicate the fork and join points, respectively.

Example. Figure 2a illustrates a piece of memory safety enforced code where the instructions with black and red colors indicate the original code and the dynamic checking code inserted, respectively. Assume that q is a pointer variable whose meta-data has been recorded beforehand. For each pointer variable defined (p or r), a meta-operation *_update* (Line C or E) is inserted accordingly to record the meta-data for each memory region they point to. As they are uninitialized, the meta-data is Null. For an assignment $r = q$ at Line F, a *_lookup* is inserted to firstly retrieve the meta-data of q, followed by an *_update* which updates the meta-data of r as that of q. Given a memory load statement $p = *r$ (Line K), as there is a dereference to the pointer r, we first need to guarantee that r is accessed legally. To this end, a *_lookup* (Line I) followed by a *_check* (Line J) is inserted. The *_lookup* gets the meta-data (*i.e.*, *md2*) of the memory region to which r points by retrieving the value associated with the address of r (*i.e.*, $\&r$) in the table. Then, the *_check* validates if the current value of r resides in the legal range and status represented by the meta-data. If failed, a runtime error will be reported. Since p is re-assigned as $*r$, its meta-data should be updated as the same meta-data of $*r$ accordingly. To this end, the *_lookup* (Line L) uses the address of $*r$ (*i.e.*, *&(*r)*, for simplicity, we directly use r) to obtain the meta-data (*i.e.*, *md3*). The subsequent *_update* (Line M) updates the meta-data of p as *md3*. Finally, for a unary dereference at Line Q, a *_lookup* first acquires the meta-data of p, followed by a *_check* that is utilized to check the validity of accessing p.

Note that different approaches [9, 12, 31] may be different concerning the implementation details, *i.e.*, the content and data structure of meta-data, the implementation of meta-operations, or the representation of programs. However, there is no doubt that all these dynamic approaches need to heavily store, search, and update the additional meta-data at runtime, which inevitably results in prohibitively high overhead. In this paper, we would like to lower the overhead by running certain meta-operations inserted in parallel.

2.2 Overview of Catamaran

Catamaran takes as input a program with a series of meta-operations instrumented, identifies the potential meta-operations which can be run in parallel, finally generates a parallel program whose overhead is greatly reduced while without sacrificing any safety guarantee.

Having the memory safety code, Catamaran first automatically discovers the potential parallelism within each memory access point. Considering the code shown as Figure 2a, there are in total five memory access points including Lines B, D, F, K, and Q. For the pointer definition points (*i.e.*, Lines B and D), their subsequent meta-operations *_update* cannot be executed in parallel due to the data dependence restriction with them. For the assignment point (Line F), as both *_lookup* (Line G) and *_update* (Line H) are data/control independent of Line F, Lines G and H can be executed in parallel with F. Similarly, for Line K, all the meta-operations before (*i.e.*, Lines I and J) it and after (*i.e.*, Lines L and M) it can be run in parallel. The same handling works for Line Q as well. After the above processing, the execution flow of the memory safety code can be demonstrated in Figure 2b, where the solid and dotted lines indicate the main thread and forked thread, respectively. The blue dot indicates the fork point, whereas the red dot corresponds to a join point. Note that the execution of instructions after a join point must wait for the end of the execution of all previous instructions.

Only the intra-point parallelism (*i.e.*, the parallel execution within a single memory access point) is insufficient. The meta-operations across different memory access points (*i.e.*, inter-point) are also possible to be run in parallel. As a result, Catamaran next performs a static dependence analysis (a.k.a., def-use analysis) to realize more parallelism across multiple memory access points on the base of the intra-point parallel tasks (*i.e.*, Figure 2b). For instance, Line C is independent of all the code (*i.e.*, Lines D-F) before L in Figure 2b. Hence, the join point for C can be pushed forward to the point right before L. As no dependence exists between L and H, the execution of L does not need to wait for the execution end of H. But there is a dependence between H and I, H has to join right before I. As shown in Figure 2c, the join point of H remains unchanged. On the contrary, the joint point following I and J can be moved down to the end of the loop (*i.e.*, Lines N-Q) as the execution of the loop is independent of Lines I and J, *i.e.*, I and J can be executed in parallel with the loop. We will give the rigorous descriptions of both intra- and inter-point task dispatch shortly in §4.

After both intra- and inter-point parallel task dispatching, a fine-grained parallel version of the memory safety code can be generated. However, too many tiny tasks could be introduced. Offloading a tiny task to a separate thread may not get any performance benefit especially when the time cost introduced by parallelism implementation is beyond the execution time of the tiny task. To this end, Catamaran exploits a profit analysis-based optimization to seek more performance gains by merging, rescheduling tasks, and handling loops, etc. For example, as shown by Figure 2d, E can be merged together with D. Meanwhile, G, H, I, and J can be unified as a single task. Moreover, instead of creating a separate thread executing O and P within each loop iteration, Catamaran divides all the meta-operations as a separate loop and instantiates a single thread for the loop. More details about parallel task optimization will be presented in §4.3.

3 PROBLEM FORMULATION

To prevent any influence on the correctness of the original program's logic, Catamaran mainly focuses on parallel acceleration of dynamic checking code and keeps the original code unchanged.

Informally, the essential problem Catamaran needs to address can be described as follows: ***given an enforced program, how do we generate a parallel program by scheduling the meta-operations such that they can be executed in parallel as much as possible while strictly preserving the happens-before relation among the code having data/control dependence?*** Catamaran adopts task-level parallelism. The execution of one or more meta-operations corresponds to one parallel task, which can be assigned to a thread for execution.

Definition 1 (**Parallel Task**). *A parallel task T is a triple $T = (\widehat{m}, l_{start}, l_{end})$ where:*

- *$\widehat{m}$ is a sequence of meta-operations which are executed sequentially by a thread.*
- *l_{start} is the position (instruction point) in the program where this task can start.*
- *l_{end} is the end position where this task must finish.*

Definition 2 (**Parallel Task Graph**). *Given a memory safety enforced program $P = \langle O, M \rangle$ where O and M denote the original code and the dynamic checking code, respectively. A parallel task graph $\mathcal{G} = \langle O, \Psi \rangle$ is the control flow graph of O augmented with a set of parallel tasks $\Psi = \{T_1, T_2, ..., T_i, ..., T_n\}$ such that the following conditions hold.*

- $\forall 1 \le i < j \le n, T_i.\widehat{m} \cap T_j.\widehat{m} = \emptyset;$
- $\bigcup_{i=1}^{n} T_i.\widehat{m} = M;$

Informally, given a control flow graph of O and a set of parallel tasks, the parallel task graph can be directly generated by embedding each parallel task into the respective position based on its start and end points. For example, Figure 2b shows a parallel task graph of the memory safety program in Figure 2a. The meta-operations of different parallel tasks are non-overlapped. While the union equals to all the meta-operations in Figure 2a. Now the problem can be formulated as follows.

Definition 3 (**Problem Statement**). *Given a memory safety enforced program $P = \langle O, M \rangle$ where $O = \{o_1, o_2, ..., o_i, ..., o_p\}$ denotes the original code and $M = \{m_1, m_2, ..., m_i, ..., m_q\}$ denotes the meta-operations inserted, the problem is equivalent to constructing a parallel task graph $\mathcal{G} = \langle O, \Psi \rangle$ such that the following conditions hold, where $\Psi = \{T_1, T_2, ..., T_i, ..., T_n\}$ represents a set of parallel tasks, and $\preccurlyeq$ indicates the happens-before relation between two control/data dependent instructions.*

- *$\forall i, j, i \in [1, p], j \in [1, q]$, if $o_i \preccurlyeq m_j$, then $o_i \preccurlyeq T_k$ where $m_j \in T_k.\widehat{m}$;*
- *$\forall i, j, 1 \le i < j \le q$, if $m_i \preccurlyeq m_j$, then $T_x \preccurlyeq T_y$ where $m_i \in T_x.\widehat{m}$ and $m_j \in T_y.\widehat{m}$;*
- *profit $(\Psi) = Cost(P) - Cost(\mathcal{G})$ is as large as possible, where $Cost$ indicates the program's execution time.*

Example. Having the example program shown as Figure 2a, the problem is essential to generate a parallel task graph shown as Figure 2d such that the profit from parallelism is as large as possible, while preserving the happens-before relation between two control/data dependent instructions.

4 APPROACH

In fact, the above optimization problem is NP-Hard meaning that it is impractical to verify whether a given parallel task graph gains the maximum profit in polynomial time. To simplify the problem, Catamaran considers each function one by one, and generates an intra-procedural parallel task graph for each function separately.

Algorithm 1: Parallel task generation

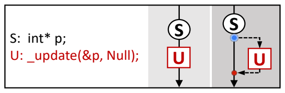
```
Input: A memory safety enforced program $\mathcal{P}$
Output: A parallel task graph $\mathcal{G}_i$ for each function $F_i$ of $\mathcal{P}$
1 foreach $F_i$ of $\mathcal{P}$ do
2     $\mathcal{G}_i \leftarrow$ IntraTaskDispatch($F_i$)
3     InterTaskDispatch($\mathcal{G}_i$, $F_i$)
4     TaskOptimization($\mathcal{G}_i$)
```

Algorithm 1 gives the pseudocode of our approach. Catamaran processes each function F_i of P separately. For a given function F_i, Catamaran first performs the intra-point parallel task dispatch that identifies the potential parallel tasks within each memory access point so as to produce an intra-point parallel task graph $\mathcal{G}_i$ (Line 2). Next, Catamaran further enlarges at most the potential range of each parallel task specified earlier without violating the happens-before visibility guarantee on the basis of a sound data/control dependence analysis (Line 3). Finally, Catamaran optimizes inter-point parallel tasks by re-scheduling certain tasks so as to pursue as much parallel profit as possible (Line 4). All the three core steps are elaborated in the following.

4.1 Intra-Point Parallel Task Dispatch

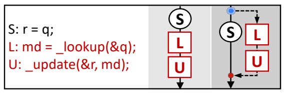
To ensure memory safety, state-of-the-art dynamic approaches usually instrument a series of dynamic checking code to track and verify memory related meta-data at the specific vulnerable memory access points. There are six types of basic pointer accesses, including pointer definition, assignment, allocation, deallocation, load, and store. For each access point, the pattern of where and what meta-operations are instrumented is fixed. Given a memory safety enforced program, at the first place, Catamaran analyzes it and automatically realizes the parallel task (the meta-operations and the corresponding start and end points) within each memory access point.

Figure 3 lists the code example, the sequential task representation, and the corresponding intra-point parallel task representation for each of the six basic pointer operations. The black circle represents an original pointer operation. Each red square indicates a



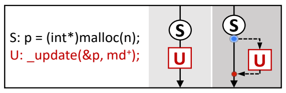
```
S:  int* p;
U: _update(&p, Null);
```

(a) pointer definition



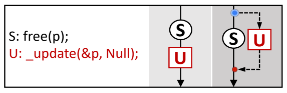
```
S: r = q;
L: md = _lookup(&q);
U: _update(&r, md);
```

(b) pointer assignment



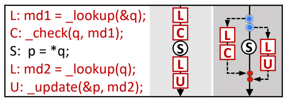
```
S: p = (int*)malloc(n);
U: _update(&p, md⁺);
```

(c) pointer allocation



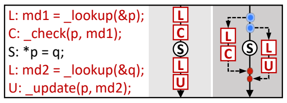
```
S: free(p);
U: _update(&p, Null);
```

(d) pointer deallocation



```
L: md1 = _lookup(&q);
C: _check(q, md1);
S:  p = *q;
L: md2 = _lookup(q);
U: _update(&p, md2);
```

(e) pointer load



```
L: md1 = _lookup(&p);
C: _check(p, md1);
S: *p = q;
L: md2 = _lookup(&q);
U: _update(p, md2);
```

(f) pointer store

Figure 3: Intra parallel tasks for six types of operations.

meta-operation inserted, where U, L, and C denote *_update*, *_lookup*, and *_check*, respectively. For example, in Figure 3f, L and C do not have happens-before data dependency with S, as well as the subsequent meta-operations L and U. Note that although C and U take the same variable p as parameter, both of them only read p. U updates the meta-data mapped by p instead of updating the value of p. Therefore, there is no data dependency between them. As such, L and C can be dispatched as an intra-point parallel task to be executed in parallel with S and the subsequent L and U. The subsequent meta-operations L and U are dispatched as an intra-point parallel task for a similar reason.

As can be seen, most of the meta-operations within each access point can be scheduled as a separate task running in parallel with the main thread. Given an enforced program, the intra-point task dispatch divides all the meta-operations into multiple parallel tasks whose start and end points are set locally as shown in Figure 3, and produces the intra-point parallel task graph.

4.2 Inter-Point Parallel Task Dispatch

As for a parallel task, it is crucial to determine where to set its start and end points. If the pair is positioned narrowly, there is not much parallelism among tasks. In fact, the larger the range is, the more parallelism can be achieved. Although intra-point task dispatch identifies many potential parallel tasks, the start and end points for each parallel task are set conservatively. In other words, it only discovers the parallelism locally within a single access point and ignores the potential parallelism across different points.

To further explore the parallelism, we would like to enlarge the start-end range for each parallel task across multiple points. However, to guarantee the correctness, the happens-before relations among all the operations on the shared variables have to be preserved. As a result, Catamaran leverages a static data/control dependence analysis to determine the largest start-end range allowed for each parallel task, while avoiding any data race. Algorithm 2 illustrates that how Catamaran dispatches parallel tasks across multiple points for a function F. Given a parallel task graph $\mathcal{G}$ of function F, each node in $\mathcal{G}$ is traversed in topological order (Line 1). For a node v, if v is a parallel task (Line 2), then we check all its direct and indirect successors v' to see if v and v' satisfy certain dependence relation. If so, v' is inserted into a set S (Line 5). After the traversal, if S is empty, the end point of v is directly set as the function F end point (Line 6); otherwise, we find the farthest common dominator of v's in S to v, and set it as the end point of v (Line 7).

Algorithm 2: Inter-point parallel task dispatch

```
Data: An intra-point parallel task graph $\mathcal{G}$ of function $F$
Result: An updated $\mathcal{G}$
1 foreach node $v \in \mathcal{G}$ do /*traverse $\mathcal{G}$ in topological order*/
2     if $v$ is a parallel task then
3         $S \leftarrow \{\}$
4         foreach direct and indirect successor $v'$ of $v$ in $\mathcal{G}$ do
5             if IsDependent($v, v'$) then $S \leftarrow S \cup \{v'\}$
6         if $S \equiv \emptyset$ then $v.l_{end} \leftarrow$ the end point of $F$
7         else $v.l_{end} \leftarrow$ FindFarthestCommonDominator($S$)
```

$$[R.1] \quad \frac{m_1 : \_lookup(k_1) \quad m_2 : \_update(k_2, \ldots) \quad k_1 \equiv k_2}{(m_1, m_2) \in D_{WAR}}$$

$$[R.2] \quad \frac{m_1 : \_update(k_1, \ldots) \quad m_2 : \_lookup(k_2) \quad k_1 \equiv k_2}{(m_1, m_2) \in D_{RAW}}$$

$$[R.3] \quad \frac{m_1 : \_update(k_1, \ldots) \quad m_2 : \_update(k_2, \ldots) \quad k_1 \equiv k_2}{(m_1, m_2) \in D_{WAW}}$$

$$[R.4] \quad \frac{o : k_1 = \ldots \quad m : \_lookup(k_2, \ldots) \quad k_1 \equiv k_2}{(o, m) \in D_{RAW}}$$

$$[R.5] \quad \frac{o : k_1 = \ldots \quad m : \_update(k_2, \ldots) \quad k_1 \equiv k_2}{(o, m) \in D_{WAW}}$$

$$[R.6] \quad \frac{o : k_1 = \ldots \quad m : \_check(k_2, \ldots) \quad k_1 \equiv k_2}{(o, m) \in D_{RAW}}$$

At Line 5 of Algorithm 2, we need to check if v' is control dependent of v or v and v' have the specific data dependence. Here we consider three types of data dependence: namely Read-After-Write (RAW), Write-After-Write (WAW), and Write-After-Read (WAR). Given that v is a parallel task, v' can be either a parallel task or an original code instruction. If v' is a parallel task, we need to determine if there exist a meta-operation $m' \in v'.\widehat{m}$ and another meta-operation $m \in v.\widehat{m}$ such that m' and m possess one of the three relations (*i.e.*, RAW, WAW, WAR). Similarly, if v' corresponds to an original instruction o, we need to check if o and the meta-operations in $v.\widehat{m}$ have one of the three data dependence relations. We treat meta-operations as white-box for alias analysis. To improve analysis efficiency, we manually specify the alias model for each meta-operation (*_update*, *_lookup*, and *_check*). The specific rules for determining the three relations are shown as R.1-R.6 where m indicates a meta-operation, o denotes an original code instruction, and $k_1 \equiv k_2$ means that k_1 and k_2 are the same variables or aliases. For example, in the R.3 rule, m_1 and m_2 update the meta-data mapped by k_1 and k_2 respectively. When k_1 and k_2 are the same variables or aliases, they update the same meta-data. It is necessary to keep the execution order of m_1 and m_2 to avoid inconsistent meta-data results. Thus, there is a WAW data dependency between m_1 and m_2, and the inter-point parallel tasks have to preserve the happens-before relation between them to avoid any incorrectness. We adopt a sound alias analysis [4] to resolve all the potential aliases of pointer variables of interest.

4.3 Parallel Task Optimization

After the intra- and inter-point parallel task dispatch, a parallel task graph with fine-grained parallel task scheduling is produced. Considering the specific amount and granularity of parallel tasks, there is still room for improvement. On the one hand, too many small parallel tasks introduce too frequent thread context switching, resulting in excessive overhead. On the other hand, too large but few parallel tasks may not take full advantage of parallel resources. To achieve more benefits, we propose to further optimize the parallel task graph generated based on a quantitative profit analysis. Catamaran manages the parallel tasks by grouping and rescheduling them so as to obtain the largest profit for the entire program. Catamaran adopts different optimizations to the loops and loop-free code, which will be discussed separately in the following.

Loop-Free Parallel Task Optimization. In the inter-point parallel task graph generated, each parallel task limits the scope for its start and end points. We can adjust each parallel task by moving its start and end points within this scope. Specifically, for a given parallel task $T = (\widehat{m}, l_{start}, l_{end})$, a series of variants $T' = (\widehat{m'}, l'_{start}, l'_{end})$ can be enumerated where $\widehat{m'} \equiv \widehat{m}$ and $l_{start} \leq l'_{start} < l'_{end} \leq l_{end}$. Moreover, two successive parallel tasks enumerated can be soundly merged as one.

Algorithm 3: Merging of inter-point parallel tasks

```
Data: Two inter-point parallel tasks $T_1$ and $T_2$, the original code of
      $T_1$ and $T_2$ spanned $O_{T_1}$ and $O_{T_2}$
Result: One merged inter-point parallel task $T$
1  if GetSucc($T_1.l_{end}$) == $T_2.l_{start}$ then
2      if HasDependency($T_2.\widehat{m}$, $O_{T_1}$) then
3          $T.l_{start} = T_2.l_{start}$
4          if HasDependency($T_1.\widehat{m}$, $O_{T_2}$) then return NULL
5          else $T.l_{end} = T_2.l_{end}$
6      else
7          $T.l_{start} = T_1.l_{start}$
8          if HasDependency($T_1.\widehat{m}$, $O_{T_2}$) then $T.l_{end} = T_1.l_{end}$
9          else $T.l_{end} = T_2.l_{end}$
10     $T.\widehat{m} \leftarrow T_1.\widehat{m} \cup T_2.\widehat{m}$
11     return $T$
```

Given two potential parallel tasks enumerated, Algorithm 3 determines if and how they can be merged as a single task. Two parallel tasks T_1 and T_2 can be merged only when $T_2.l_{start}$ is the successor of $T_1.l_{end}$ (Line 1). To soundly decide the start and end points of the merged task T, we firstly check whether the meta-operations of T_2 and the original code of T_1 spanned satisfy certain dependence relation (Line 2). The HasDependency function determines if there is a dependence relation between meta-operations and original code according to the rules R.4-R.6. If so, $T.l_{start}$ is set as $T_2.l_{start}$ (Line 3), which means T_2's start point cannot be safely hoisted up. Next, we check the dependency between $T_1.\widehat{m}$ and O_{T_2} to determine the merged end point of T (Line 4). If so, there is no need to merge two tasks. Otherwise, T_1's end point can be safely moved down and $T.l_{end}$ is set as $T_2.l_{end}$(Line 5). If there is no dependency between $T_2.\widehat{m}$ and O_{T_1} (Line 6), T_2's start point can be lifted beyond T_1's start point. $T.l_{start}$ is thus set as $T_1.l_{start}$ (Line 7). Further, if there is a dependence relation between $T_1.\widehat{m}$ and O_{T_2}, T_1's end point cannot be safely moved down, then $T.l_{end}$ is set as $T_1.l_{end}$ (Line 8). If not, T's end point can be safely set as $T_2.l_{end}$(Line 9). Finally, we assign the union of T_1 and T_2's meta-operations to $T.\widehat{m}$ (Line 10) and return the merged parallel task T (Line 11).

By enumerating all the parallel tasks, a large number of valid parallel task graphs can be generated, in the sense that all these generated graphs satisfy the first two conditions specified in the problem statement (definition 3). What we should do next is to select from all the valid task graphs one which owns the largest profit. To this end, we quantitatively estimate the execution cost of each given parallel task graph based on a cost model discussed shortly. By computing and comparing the costs of all the valid parallel task graphs, we can identify one from them which has the smallest cost.

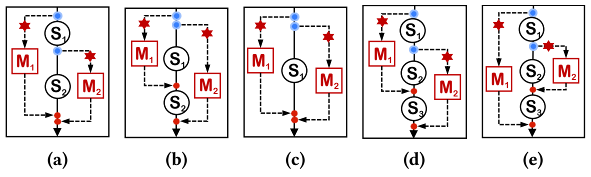


Figure 4: Five basic patterns of parallel tasks.

A cost model for loop-free parallel tasks is introduced as follows to estimate its execution cost.

Definition 4 (**Cost Model**). *In a parallel task graph, there are five basic parallel task patterns shown in Figure 4. For each pattern, its time cost can be modeled as follows where t_M indicates the cost of meta-operations in a parallel task, t_S indicates the cost of original code, and t_X represents the cost of thread implementation.*

- $C_a = \max\{t_{M_1} + t_X, t_{S_1} + \max\{t_{S_2}, t_{M_2} + t_X\}\}$
- $C_b = \max\{\max\{t_{M_1} + t_X, t_{S_1}\} + t_{S_2}, t_{M_2} + t_X\}$
- $C_c = \max\{t_{M_1} + t_X, t_{S_1}, t_{M_2} + t_X\}$
- $C_d = \max\{t_{M_1} + t_X + t_{S_3}, t_{S_1} + \max\{t_{S_2} + t_{S_3}, t_{M_2} + t_x\}\}$
- $C_e = \max\{t_{M_1} + t_x, t_{S_1} + \max\{t_{S_2}, t_{M_2} + t_x\} + t_{S_3}\}$

As the cost is needed statically, we leverage the static CPU cycles to estimate the costs represented by each parameter in the above model. To calculate the actual value of each parameter in the cost model, we adopt both static CPU cycle-based and dynamic profiling-based modeling. To estimate t_M and t_S, we directly utilize the static analyzer supported by LLVM to get the sum of CPU cycles for M and S. As for t_X, a profiling method is applied. We randomly select several intra-point parallel tasks with the meta-operations M and original code S running in parallel with known CPU cycles of t_M and t_S under a test suite. We record the actual execution time of M (TE_M) and S (TE_S) under serial execution, and the total execution time (TE_P) under parallel execution. Then, the actual time cost for thread implementation TE_X can be calculated by the equation $TE_X = TE_P - max\{TE_M, TE_S\}$. To get the stable results, the TE_X value is computed by taking the average value of ten runs on each test. Having the actual value of TE_X, we need to map its actual value to the number of CPU cycles. To this end, we use a linear regression to simulate the mapping function with a positive correlation between CPU cycles and the actual execution time on a set of tests.

Loop Parallel Task Optimization. Catamaran handles loops in a different way from that of loop-free code. For example, there is a loop with meta-operations O and P being inserted in the body, shown as Lines N-Q in Figure 2a. After the inter-point parallel task dispatch, a parallel task executing O and P is created within each loop iteration as illustrated by Figure 2c. Suppose that at one iteration of the loop, the execution cost required by meta-operations O and P is t_{OP} (*i.e.*, t_M in definition 4); the cost executed by the original code Q is t_Q (*i.e.*, t_S in definition 4); t_X represents the cost of thread implementation and the number of loop iteration is N. According to the cost model shown as definition 4, we can calculate the cost of original safety enforced code and that after inter parallel task dispatch as $C_{original}$ and C_{inter} as follows, respectively.

$$\begin{cases} C_{original} = N \times (t_{OP} + t_Q) \\ C_{inter} = N \times max\{t_Q, t_{OP} + t_X\} \end{cases} \tag{1}$$

For tiny loops where the loop body only contains a few instructions, t_Q is usually much smaller than t_{OP}, sometimes even comparable to t_X. In this case, the following equation holds.

$$C_{inter} \approx N \times (t_{OP} + t_X) \tag{2}$$

As such, we can hardly achieve any performance gain via inter-point parallel task dispatch. Even worse, it may probably introduce extra overhead to the original safety code especially when t_X is larger than t_Q. In other words, instantiating small parallel tasks in a loop not only gets little performance gain, but causes more overhead. To optimize it, Catamaran extracts a separate loop with only the meta-operations inside, and treats the whole loop as a parallel task. In essence, this is equivalent to unrolling the loop infinitely and then applying the above loop-free parallel task optimization to it. We term this special parallel task as loop parallel task. The start point of the loop parallel task is set conservatively as the pre-header position of the original loop. The end point is placed at the end of the function for more parallelism. In this way, the total cost C_{opt} of the loop optimized can be calculated as follows. It can be easily seen from eq. (2) and eq. (3) that C_{opt} has $(N-1) \times t_X$ fewer CPU cycles than C_{inter}. Therefore, loop parallel tasks can eliminate the redundant thread instantiation overhead, while still enjoying the benefits of parallelism.

$$C_{opt} = N \times max\{t_Q, t_{OP}\} + t_X \approx N \times t_{OP} + t_X \tag{3}$$

We also provide load balance optimization for loop parallel tasks. According to the cost model (definition 4), the loop code cost is similarly dominated by the maximum between the t_M and t_S. Thus, we try the best to adjust and balance the costs of the two loops (*i.e.*, the original loop and the loop parallel task extracted), which supports for gaining more parallel profits. For the case where t_M is much greater than t_S, instead of extracting all the meta-operations into the loop parallel task, we only separate a portion of them so as to make the cost of two loops as close as possible.

For nested loops, Catamaran only optimizes the outermost loop at the current moment. Apparently, in most cases, to make the extracted loop run successfully, certain original code in the loop perhaps needs to be re-executed in the loop parallel task. To avoid data race and false sharing problems, Catamaran allocates an extra memory space to maintain certain local variables (*e.g.*, loop index), while initializing them and the corresponding meta-data if necessary. Note that not all the loops can be optimized. We cannot do any optimization under the following cases, *e.g.*, the loop contains thread-safe functions (*e.g.*, *malloc*, *free*, *scanf*) which are implemented using mutex; a function pointer or recursion is included in a loop. For the loops that cannot be specially optimized, Catamaran will optimize it as loop-free code.

5 IMPLEMENTATION

Catamaran is implemented as an optimization pass of LLVM. It takes as input a memory safe program in the form of LLVM IR, and eventually generates a parallel version.

Parallel Task Dispatch. Catamaran analyzes def/use relations benefiting from the SSA form of LLVM IR by following [14], and exploits the existing pointer analysis [4] to detect all the alias information. Moreover, Catamaran leverages the Semi-NCA algorithm [15] which is provided by the LLVM (Post-)DominatorTree pass to discover all (post-)dominator nodes of interest.

Parallel Task Optimization. Catamaran exploits the cost model of TTI [2] in LLVM to estimate t_M and t_S. To obtain t_X, dynamic profiling is done once for each baseline tool under each library setup. Catamaran uses the LoopInfo pass [1] in LLVM to identify loop code, and refers to NOELLE [27] to implement loop transformation. Note that, to extract a separate loop, Catamaran allocates separate copies of all live variables (together with their corresponding meta-data if needed) used for the loop execution.

Parallel Code Transformation. Catamaran creates a wrapper function for each optimized parallel task T, takes the necessary program values as function parameters to be passed, and completes the memory allocation and initialization. Catamaran inserts the thread create and join function at the l_{start} and l_{end} points with an unsigned parallel task ID randomly assigned. An effective thread pool with work stealing algorithm enabled is designed and implemented to provide multi-threaded runtime for manipulating parallel tasks efficiently. Considering the unsound impact of thread-unsafe meta-operations in SoftBoundCETS and MoveC, we have to avoid data races when accessing the shared lookup table in parallel. To this end, we modify the original runtime libraries and provide the thread-safe variants. Meanwhile, we only add fine-grained locks to the innermost meta-data entry for meta-operations, thus lowering the synchronization overhead as much as possible.

6 EVALUATION

The goal of Catamaran is to reduce the runtime overhead of dynamic memory safety enforcement via task parallelization. It is general enough to benefit a large number of existing dynamic memory safety enforcement approaches/tools [9, 12, 31, 40]. In our experiments, we selected three representative dynamic memory enforcement tools SoftBoundCETS [31], MoveC [9], and AddressSanitizer [40] (ASAN for short) as the subject tools to evaluate how well Catamaran can lower the overheads introduced by them. SoftBoundCETS and ASAN instrument meta-operation code at the IR level whereas MoveC enforces the program at the source code level. We evaluated SoftBoundCETS (for LLVM-3.4) with full checks (*i.e.*, both spatial and temporal checks) enabled, MoveC (version 0.9.0) by default setting, and ASAN (for LLVM-3.5.2) with outlined checks. All the experiments were conducted on a commodity PC with an Intel Xeon W-2145 8-Core CPU, 64GB memory, and 1T SSD, running Ubuntu 16.04.

Table 2: Overhead reduction of Catamaran for SoftBoundCETS, MoveC and ASAN on SPEC CPU2006 and 2017 benchmark under -O3 compiler optimization setting with 4 threads available; columns TO_β and TO_γ denote time overhead of β and γ versions, respectively; MO indicates memory overhead; $\triangle TO$ means $TO_\beta - TO_\gamma$, similar for $\triangle MO$; RTO means TO_β / TO_γ; Abort indicates that SoftBoundCETS detects memory violation and aborts the program; IE indicates instrumentation failure; RE denotes that the program reports runtime error (*e.g.*, segmentation fault).

Subject	SoftBoundCETS [31]							MoveC [9]							ASAN [40]						
	TO_β	TO_γ	$\triangle TO$	RTO	MO_β	MO_γ	$\triangle MO$	TO_β	TO_γ	$\triangle TO$	RTO	MO_β	MO_γ	$\triangle MO$	TO_β	TO_γ	$\triangle TO$	RTO	MO_β	MO_γ	$\triangle MO$
perlbench	Abort	Abort	-	-	-	-	-	IE	IE	-	-	-	-	-	318%	247%	71%	1.29x	240%	241%	1%
bzip2	183%	76%	107%	2.42x	132%	134%	2%	719%	302%	417%	2.38x	0.09%	0.50%	0.41%	151%	86%	65%	1.76x	4%	4.40%	0.40%
gcc	IE	IE	-	-	-	-	-	IE	IE	-	-	-	-	-	201%	173%	28%	1.16x	216%	217%	1%
mcf	324%	183%	141%	1.77x	400%	407%	7%	523%	301%	223%	1.74x	400%	411%	11%	84%	46%	38%	1.84x	13%	15%	2%
milc	132%	62%	70%	2.13x	238%	256%	18%	IE	IE	-	-	-	-	-	70%	51%	19%	1.37x	42%	46%	4%
gobmk	216%	173%	43%	1.25x	998%	1012%	14%	IE	IE	-	-	-	-	-	112%	88%	24%	1.27x	1142%	1152%	10%
hmmer	253%	134%	119%	1.89x	1336%	1350%	14%	IE	IE	-	-	-	-	-	265%	156%	109%	1.70x	1616%	1624%	8%
sjeng	153%	131%	22%	1.17x	7%	9%	2%	515%	380%	171%	1.45x	0.63%	2.85%	2.22%	133%	85%	48%	1.56x	7%	10%	3%
libquantum	Abort	Abort	-	-	-	-	-	748%	292%	456%	2.56x	0.08%	1.06%	0.98%	37%	20%	17%	1.85x	274%	280%	6%
lbm	58%	29%	29%	1.99x	0.18%	0.59%	0.41%	589%	299%	290%	1.97x	0.06%	0.51%	0.45%	66%	41%	25%	1.61x	15%	16%	1%
sphinx3	336%	215%	121%	1.56x	712%	720%	8%	RE	RE	-	-	-	-	-	128%	92%	36%	1.39x	940%	949%	9%
x264_r	Abort	Abort	-	-	-	-	-	1082%	1051%	32%	1.03x	12%	17%	5%	179%	114%	65%	1.57x	27%	31%	4%
imagick_r	Abort	Abort	-	-	-	-	-	326%	296%	30%	1.10x	88%	93%	5%	184%	110%	74%	1.68x	188%	189%	1%
nab_r	100%	47%	53%	2.15x	157%	165%	8%	IE	IE	-	-	-	-	-	86%	48%	38%	1.81x	272%	274%	2%
xz_r	260%	196%	65%	1.33x	13%	14%	1%	781%	610%	171%	1.28x	0.37%	1.04%	0.67%	94%	65%	29%	1.45x	32%	33%	1%
Mean			77%	1.77x			7.44%			224%	1.69x			3.22%			46%	1.55x			3.56%

We intend to answer the following research questions:

- Q1: How well does Catamaran reduce the runtime overheads suffered by SoftBoundCETS, MoveC, and ASAN? (§6.1)
- Q2: How does Catamaran perform compared to the state-of-the-art? (§6.2)
- Q3: What about the impact of the number of threads available and the compiler's optimizations on Catamaran's performance? (§6.3)
- Q4: Does Catamaran affect the capability of detecting memory errors? (§6.4)

6.1 Performance

In the performance experiments, we chose the commonly used SPEC CPU 2006 and 2017 benchmarks. Excluding the subjects written in Fortran or C++ that could not be instrumented by SoftBoundCETS and MoveC, we ended up retaining 15 C subjects. The first column of Table 2 lists all the subjects used. The last four subjects with suffix _r belong to SPEC CPU 2017. Each subject comes with a training workload used for cost model profiling, and a reference workload used for measuring overhead. We compiled for each subject a baseline version without any instrumentation (α), a memory safety enforced version (β) produced by SoftBoundCETS, MoveC, or ASAN, and a corresponding version Catamaran parallelized (γ). Unfortunately, not all subjects are compatible with the reference tools; SoftBoundCETS succeeds in processing 10 subjects and MoveC can handle 8 subjects under -O3 compiler optimization setting. The rest of subjects failed due to either compilation errors or instrumentation errors. We collected the average execution elapsed time and peak memory used by running each version ten times, and calculated both time and memory overhead of them.

Table 2 shows the performance of Catamaran under -O3 compiler optimization setting with 4 threads available. The columns TO_β and TO_γ denote the time overhead of β and γ versions, respectively. MO indicates the memory overhead. $\triangle TO$ means $TO_\beta - TO_\gamma$, similar for $\triangle MO$. RTO means the ratio, *i.e.*, TO_β / TO_γ. **Abort** indicates SoftBoundCETS detects the memory violation and aborts the program. **IE** indicates the instrumentation failure. **RE** denotes that the program reports runtime error (*e.g.*, segmentation fault). Note that for MoveC, we keep the default setting – when a memory violation is detected, it continues the execution while reporting the error information to the shell. All the β versions in Table 2 are running on top of the original thread-unsafe runtime libraries.

As can be seen, Catamaran is able to effectively reduce the overhead introduced by SoftBoundCETS, MoveC, and ASAN. For subjects under SoftBoundCETS instrumentation, Catamaran removes 22%-141% of overhead. Especially, Catamaran has a significant effect on removing the overhead for *bzip2* where the ratio TO_β / TO_γ is up to 2.42x, while 2.15x and 2.13x for *nab_r* and *milc* respectively. For subjects under MoveC instrumentation, 30%-456% of overhead can be decreased by Catamaran. In particular, the overhead reduction ratio on *libquantum* reaches 2.56x and on *bzip2* reaches 2.38x. For ASAN[1], Catamaran also achieves the speedup ratio of 1.55x on average. As the meta-operations of updating and looking up in ASAN are directly inlined into the original code, Catamaran is not able to parallelize these meta-operations. Meanwhile, the memory overhead introduced by Catamaran is negligible where it mainly comes from the creation of the thread pool and the allocation of additional memory for executing loop parallel tasks. Only 7.44% for SoftBoundCETS, 3.22% for MoveC, and 3.56% for ASAN extra memory overhead is introduced on average, and nearly half of subjects introduce less than 3% memory overhead. A few subjects such as *milc* and *gobmk*, have slightly higher memory overheads due to slightly more variable copies allocated in the loop parallel tasks.

However, the acceleration of Catamaran over some subjects, such as *gobmk*, *sjeng*, *x264_r*, and *imagick_r*, is limited. For *gobmk*, there are a large number of mutex-locked library functions involving I/O

[1]Note that the average overhead of ASAN in Table 2 is slightly higher than that reported in [40], since we outlined checks in ASAN (supported from LLVM-3.5.2) for adaption to Catamaran.

operations, which Catamaran can seldom accelerate via parallelization. As for *sjeng*, plenty of recursive functions are included. Hence Catamaran cannot process them, resulting in fewer benefits. For *x264_r* and *imagick_r* enforced by MoveC, as intensive logging is constantly exported during running since MoveC continues the execution for detecting multiple errors in a run, the parallel profit cannot be reflected in the elapsed time.

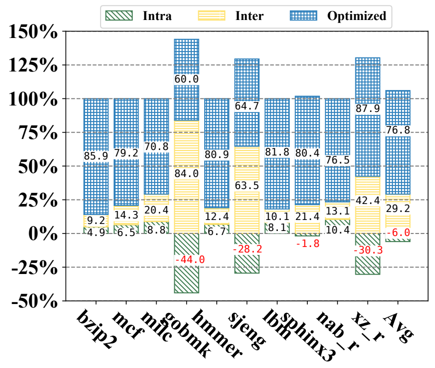


Figure 5: Percentage of each step contributed to the overall overhead reduction for SoftBoundCETS.

Step-by-Step Performance Improvement. To understand the contributions of each step of Catamaran (*i.e.*, *intra dispatch* in §4.1, *inter dispatch* in §4.2, *task optimization* in §4.3) to the overall performance, we conducted an ablation study. We limited the parallelization processing step by step and generated the respective parallel binary code accordingly. Specifically, the intra step ends after the execution of Line 2 in Algorithm 1 and produces the corresponding parallelized binary code (γ-intra). Similarly, the inter step generates binary code (γ-inter) after Line 3 in Algorithm 1 is executed. The optimization step generates the code (γ-opt) after the whole process of Algorithm 1, which is identical to the version γ. We run each binary code γ-intra, γ-inter, and γ-opt to collect the overhead $TO_{\gamma\text{-intra}}$, $TO_{\gamma\text{-inter}}$, and $TO_{\gamma\text{-opt}}$, respectively. The percentage of each step contributed is computed as $\text{Percent}_{\text{intra}} = (TO_\beta - TO_{\gamma\text{-intra}})/(TO_\beta - TO_\gamma)$, $\text{Percent}_{\text{inter}} = (TO_{\gamma\text{-intra}} - TO_{\gamma\text{-inter}})/(TO_\beta - TO_\gamma)$, and $\text{Percent}_{\text{opt}} = (TO_{\gamma\text{-inter}} - TO_{\gamma\text{-opt}})/(TO_\beta - TO_\gamma)$.

Figure 5 shows the percentage of each step contributed to the overall overhead reduction for SoftBoundCETS. Note that the intra step (*i.e.*, **Intra**) may bring negative impact meaning that the overhead $TO_{\gamma\text{-intra}}$ is even larger than TO_β, resulting in negative values. As a result, the y-axis values possibly exceed 100%, but the percentage sum of three steps must equal to 100% on each subject. It can be seen that the contribution of the intra step is negligible on somes subjects for SoftBoundCETS. The average value is -6.0%. The main reason is that the intra step only considers limited parallelism within a single memory access point. Moreover, the cost paid due to the intensive thread assignments and joins is even higher than the profit gained from parallelization. With more parallelism exploited in a broader scope, the inter step (*i.e.*, **Inter**) shows better performance than that of the intra step. However, its gain is still limited and the average contribution percentage is only 29.2% for SoftBoundCETS. In the optimization step (*i.e.*, **Optimized**) where specific parallel task merging/rescheduling are considered, significant overhead reductions are accomplished. On average, the optimization step contributes 76.8% for SoftBoundCETS to the overall performance gain. Note that the similar step-by-step contribution distribution is observed in MoveC and ASAN.

Table 3: Overhead reduction of Catamaran and Reference [24] for SoftBoundCETS.

Subject	TO_β	Catamaran			Reference [24]		
		TO_γ	$\triangle TO$	RTO	TO_γ	$\triangle TO$	RTO
bzip2	183%	76%	107%	2.42x	145%	38%	1.26x
mcf	324%	183%	141%	1.77x	249%	75%	1.30x
milc	132%	62%	70%	2.13x	80%	52%	1.65x
gobmk	216%	173%	43%	1.25x	194%	22%	1.11x
hmmer	253%	134%	119%	1.89x	235%	18%	1.08x
sjeng	153%	131%	22%	1.17x	137%	16%	1.12x
lbm	58%	29%	29%	1.99x	51%	7%	1.14x
sphinx3	336%	215%	121%	1.56x	334%	2%	1.01x
nab_r	100%	47%	53%	2.15x	88%	12%	1.14x
xz_r	260%	196%	65%	1.33x	217%	43%	1.20x
Mean			77%	1.77x		28%	1.20x

6.2 Comparison against the State-of-the-Art

To verify how well Catamaran performs compared to the state-of-the-art, we would like to conduct the empirical comparisons between Catamaran and other decoupling/parallelization approaches [24, 33, 36]. Unfortunately, none of the existing approaches' implementations is publicly accessible. Since no existing implementations for direct comparison, we implemented by ourselves a prototype of [24] which is the most recent and related work to Catamaran, by faithfully following the key insight and algorithm described in their paper. Reference [24] extracts separate meta-functions/operations from the original code and executes them in customized helper threads. The helper threads communicate with the main thread via pipelined communication, which passes the required data to perform the meta-operations. We selected SoftBoundCETS as the candidate tool to be accelerated as its meta-operations fit best that defined in [24]. We conducted the experiments on both Reference [24] and Catamaran under the same setting as that of Table 2. Table 3 shows the comparison results in terms of overhead reduction.

We can read from Table 3 that Catamaran reduces much more overheads on all subjects. On average, Catamaran is able to eliminate around 77% runtime overheads, whereas Reference [24] reduces 28%. The reasons why Catamaran outperforms [24] are threefold. First, the meta-operations which can be parallelized [24] are statically determined and manually specified a priori. In such case, only the spatial *_check* and *_update* statically satisfying the dependence restriction can be parallelized, leading to the limited number of parallel tasks. Second, [24] allows only one helper thread to execute in parallel with the main thread at runtime, meaning that only the parallelism between the meta-operation and original code is exploited. Finally, optimizations like merging and rescheduling parallel tasks are not considered in [24]. In a word, Catamaran outperforms [24] thanks to the holistic design including dynamic identification of potential parallel tasks, parallelism exploitation among meta-operations themselves, and profit-based optimizations.

6.3 Effects of Thread Count and Compiler's Optimizations

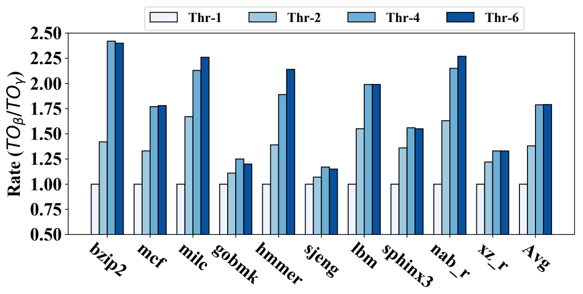


Figure 6: Overhead ratio under different numbers of threads (1, 2, 4, 6) for SoftBoundCETS.

Thread Count. To understand the impact of thread count, we collected the performance data of Catamaran under different numbers of threads available. Specifically, we adjusted the maximum number of threads in the thread pool to 2, 4, and 6. Based on the time overhead collected, we computed the ratio (*i.e.*, TO_β/TO_γ) for each setting. Figure 6 shows the ratio of time overhead under different numbers of threads for SoftBoundCETS. Thr-1 corresponds to the baseline where no overhead is reduced (*i.e.*, Catamaran is not applicable as no additional thread is available). What stands out in Figure 6 is that the increase of thread count results in more overhead reduction for most subjects as more threads enables more tasks to execute in parallel. This trend is significant especially for the numbers 1, 2, and 4. However, with 6 threads available, the reduction ratios of certain subjects (*e.g.*, *gobmk*, *sjeng*, *lbm*, and *sphinx3*) are not improved compared with that of 4. Instead of bringing profit, the oversupplied threads introduce extra overhead, which degrades the overall performance. Moreover, the average memory overhead is 3.58%, 7.44%, and 7.48% under 2, 4, and 6 threads, respectively. Since more threads need more memory for computation, the peak memory becomes slightly larger with the increase of threads. MoveC and ASAN show the same trends in terms of reduction ratio and memory overhead.

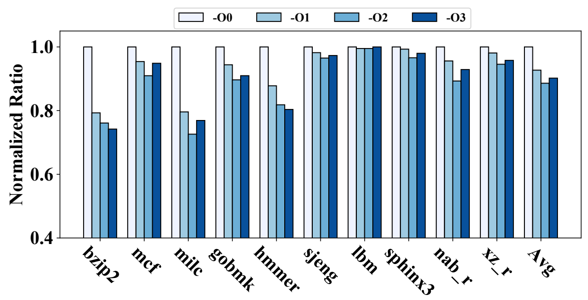


Figure 7: Normalized overhead ratio under different compiler optimization levels (-O0, -O1, -O2, -O3) for SoftBoundCETS.

Compiler's Optimization Level. To understand the effect of compiler optimization levels, we produce three versions of code (*i.e.*, α, β, and γ) under each compiler setting (*i.e.*, -O0, -O1, -O2, and -O3). We measure the overhead reduction ratios of Catamaran with 4 threads available compared to SoftBoundCETS, which are demonstrated as Figure 7. For ease of presentation, we take the data of -O0 as the baseline and normalize the reduction ratios for each optimization level. For most subjects, the reduction ratio with compiler optimization enabled is lower than that without optimization. This is because the overheads introduced by SoftBoundCETS is decreased due to the optimization, meaning that the potential parallelism profit of Catamaran is also decreased. Moreover, some loops are unrolled during compiler optimization, causing the loop-specific optimization of Catamaran inapplicable. However, we also have observed that for certain subjects (*e.g.*, *sjeng* and *xz_r*), the reduction ratios are improved when more aggressive optimizations are applied, such as from -O2 to -O3. This is because the higher optimization level aggressively inlines certain functions, creating more parallelism opportunities for Catamaran. In addition, the average memory overhead is 8.25%, 7.65%, 7.31%, and 7.44% for O0, O1, O2, and O3, respectively. The similar results are observed in both MoveC and ASAN.

6.4 Effectiveness of Safety Enforcement

Apart from the overhead reduction, we also empirically validated the effectiveness of Catamaran on memory safety enforcement. In other words, we would like to measure if Catamaran degrades the ability of detecting memory errors compared to the reference tools (*i.e.*, SoftBoundCETS, MoveC, and ASAN). We firstly selected the artificial MoveC-MSBench [9], which contains three test suites (*i.e.*, *all-mem-err*, *C-syntax*, and *nptrs*) covering a rich set of memory errors. The *all-mem-err* test suite aims at checking whether a tool can detect all known types of memory errors. The *C-syntax* test suite is used to test whether a tool supports the diverse syntax of C. The *nptrs* is mainly for testing if a tool can detect spatial memory errors under extreme stress. We run both β and γ versions over the three test suites. Ultimately, we got the same detection results of two versions for SoftBoundCETS, MoveC, and ASAN.

Table 4: Real-life CVEs used for evaluation; ✓ denotes that the memory error can be successfully detected.

Software	CVE		SoftBoundCETS		ASAN	
	ID	Type	β	γ	β	γ
libzip-1.2.0	CVE-2017-12858	use after free	✓	✓	✓	✓
zziplib-0.13.62	CVE-2017-5978	buffer overflow	✓	✓	✓	✓
libtiff-4.0.7	CVE-2016-10270	buffer overflow	✓	✓	✓	✓
	CVE-2016-10271	buffer overflow	✓	✓	✓	✓
	CVE-2016-10095	buffer overflow	✓	✓	✓	✓
graphicsmagick -1.3.26	CVE-2017-12936	use after free	✓	✓	✓	✓
	CVE-2017-12937	buffer overflow	✓	✓	✓	✓
potrace-1.14	CVE-2017-7263	buffer overflow	✓	✓	✓	✓

Besides the MoveC-MSBench, we also considered the real-life CVEs used by [25, 48] from the LinuxFlaw [29] repository. We examined all the CVEs and found all which satisfies the following three conditions: 1) is memory-related error; 2) can be successfully compiled and reproduced in our environment; 3) can be successfully instrumented by SoftBoundCETS, MoveC, or ASAN. Finally, 8 real-world vulnerabilities are selected for experiments. Table 4 shows the effectiveness results of β version and γ version under 8 real-life vulnerabilities. It can be seen that γ version parallelized by Catamaran is able to detect all the 8 memory errors, which shows the identical capability to that of the β version generated by SoftBoundCETS and ASAN. Here we did not report any data of MoveC as it fails to process these programs due to instrumentation errors.

7 DISCUSSION

Correctness of Parallel Enforced Program. Catamaran ensures the correctness of the parallel enforced program from two aspects. First of all, Catamaran only parallelizes the dynamic checking code, leaving the original code to run as usual in the main thread. As all the states of the main thread are read-only to the dynamic checking code, the execution of dynamic checking code will not change any state of the main thread. As a result, scheduling the execution order of dynamic checking code will not affect the execution correctness of the original code. Moreover, the correctness of executing dynamic checking code is not affected as well since the happens-before relations among all the control/data dependent code are rigorously preserved by the sound dependence analysis. To sum up, the output parallel program generated by Catamaran retains the functionalities of both the original code and dynamic checking code. The execution correctness is guaranteed.

Applicability of Catamaran. Catamaran allows the checking meta-operation *_check* to be executed in parallel with the original dereference instruction. In other words, it is possible that the original dereference code is executed before *_check* in the parallel program. This relaxation does not affect the error detection since the checking code will be executed anyway and the same error is reported. However, it might be unsuitable for certain scenarios where the program is forced to be terminated before the memory violation happens or timely reports errors. From this point of view, Catamaran is more tailored to the offline scenarios such as testing, debugging, and development process where timely security mitigation is not the top priority. In such scenarios, it is worthwhile to sacrifice some of the timeliness of error reporting in exchange for a significant performance improvement. Moreover, Catamaran currently only considers single-threaded applications as subjects. We leave the support of multi-threaded programs as future work.

8 RELATED WORK

Dynamic Memory Safety Enforcement. Thanks to the high precision, many dynamic approaches [3, 9, 11, 12, 18–20, 31, 32, 40] have been widely adopted to enforce memory safety. However, the prohibitively high overhead severely undermines their applicability. Catamaran focuses on reducing the runtime overhead of these dynamic approaches, while without sacrificing their ability in memory safety enforcement.

Runtime Overhead Reduction. Several studies leveraged static [22, 41, 46] and/or dynamic analysis [44, 48] to remove the redundant checking code. Unfortunately, they are limited by the imprecision and/or incompleteness of program analyses. In contrast, Catamaran transforms parallel tasks on the basis of dependence analysis, which ensures that no false positives or negatives will be added. More importantly, our approach is orthogonal to the program analysis-based approaches. Catamaran can take the output of them, further reduce the overhead. Another way to reduce checking code is to add annotations manually, such as Checked C [13]. However, writing annotations is labor-intensive and error-prone. It takes plenty of time to understand the code, which is impractical for large scale programs. Moreover, there exists a group of work [30, 34, 42, 45] speeding up the execution of checking code with hardware assistance. Although the performance is significantly improved by hardware acceleration, it strongly depends on the specific hardware and compiler supports. While Catamaran can run on a commodity PC without requiring any specific supports. Another category for overhead reduction is via decoupling. [36] customized a guarded program by decoupling the checking logic from the original and launched an additional shadow process to execute it, where parallelism among dynamic checking code is not explored. Speck [33] decouples the dynamic checking code and executes it on other cores. However, due to the dependence between the original code and dynamic checking code, the original code has to be frequently replayed on other cores to produce the state required for running dynamic checking code correctly. The log-based architectures [7, 16, 43] divide dynamic checking code into different types of events. One core produces an event stream and passes it to another core via logs. However, the execution in each event stream is not parallel. Even worse, it has to endure the heavy communication cost via aggressive log passing. The most related work to Catamaran is [24], which extracts meta-operations from the original program and executes them in customized helper threads. As discussed before in §6.2, it requires manual intervention to specify the code decoupled a priori. Meanwhile, its performance is severely limited.

Automatic Parallelization. Plenty of work [5, 37–39] have been proposed to automatically parallelize the sequential code. Different from these approaches, Catamaran only parallelizes the meta-operations, leaving the original code unchanged. Several work (*e.g.*, DSWP [35], DOACROSS [10], DOALL [21]) parallelize loop structures. Catamaran can also benefit from these automatic loop parallelization paradigms, especially for loop parallel task optimization.

9 CONCLUSION

We propose a novel approach for reducing runtime overhead of memory safety enforcement via parallel acceleration. We manipulate the dynamic checking code and run them in parallel, thus lowering the time overhead. By leveraging the sound static analysis, we ensure that all the happens-before relations between the data/-control dependent code are strictly preserved, without affecting the correctness of the memory enforced program. Evaluations over a rich set of artificial and real-world programs validate that our approach is able to significantly reduce the time overhead introduced by the existing dynamic enforcement tools, while without retrograding their capability of memory error detection.

10 DATA AVAILABILITY

All the experimental data can be accessed via the link: https://figshare.com/articles/dataset/material_issta23/22099655. The artifact is publicly avaiable on Zenodo via the link: https://doi.org/10.5281/zenodo.7957261.

ACKNOWLEDGMENTS

We would like to thank all the anonymous reviewers for their valuable comments. This work is partially supported by the National Natural Science Foundation of China (Grants 62032010 and 62272217), the Fundamental Research Funds for the Central Universities (Grant 020214380104), and the Huawei-NJU Innovation Lab for Next-Generation Programming (Project YBN2019105178SW21).

REFERENCES

[1] 2022. LLVM LoopInfo Class Reference. https://llvm.org/doxygen/classllvm_1_1LoopInfo.html. Accessed: 2022-3-1.

[2] 2022. LLVM TargetTransformInfo Class Reference. https://llvm.org/doxygen/classllvm_1_1TargetTransformInfo.html. Accessed: 2022-3-1.

[3] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors.. In *USENIX Security Symposium*, Vol. 10.

[4] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph. D. Dissertation. Citeseer.

[5] Sotiris Apostolakis, Ziyang Xu, Greg Chan, Simone Campanoni, and David I August. 2020. Perspective: A sensible approach to speculative automatic parallelization. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems.* 351–367.

[6] Luiz André Barroso and Urs Hölzle. 2009. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture* 4, 1 (2009), 1–108.

[7] Shimin Chen, Babak Falsafi, Phillip B Gibbons, Michael Kozuch, Todd C Mowry, Radu Teodorescu, Anastassia Ailamaki, Limor Fix, Gregory R Ganger, Bin Lin, et al. 2006. Log-based architectures for general-purpose monitoring of deployed code. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability.* 63–65.

[8] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B Gibbons, Todd C Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. 2008. Flexible hardware acceleration for instruction-grain program monitoring. *ACM SIGARCH Computer Architecture News* 36, 3 (2008), 377–388.

[9] Zhe Chen, Junqi Yan, Shuanglong Kan, Ju Qian, and Jingling Xue. 2019. Detecting memory errors at runtime with source-level instrumentation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 341–351.

[10] Ron Cytron. 1986. Doacross: Beyond vectorization for multiprocessors. In *Proc. of the Int. Conf. on Parallel Processing, 1986.*

[11] Gregory J Duck and Roland HC Yap. 2016. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction.* 132–142.

[12] Gregory J Duck and Roland HC Yap. 2018. EffectiveSan: type and memory error detection using dynamically typed C/C++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 181–195.

[13] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. 2018. Checked C: Making C safe by extension. In *2018 IEEE Cybersecurity Development (SecDev).* IEEE, 53–60.

[14] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.

[15] Loukas Georgiadis, Renato F Werneck, Robert E Tarjan, Spyridon Triantafyllis, and David I August. 2004. Finding dominators in practice. In *European Symposium on Algorithms.* Springer, 677–688.

[16] Michelle L Goodstein, Evangelos Vlachos, Shimin Chen, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. 2010. Butterfly analysis: Adapting dataflow analysis to dynamic parallel monitoring. *ACM SIGARCH Computer Architecture News* 38, 1 (2010), 257–270.

[17] Google. 2020. The memory safety report of the Chromium project. https://www.chromium.org/Home/chromium-security/memory-safety. Accessed: 2022-3-1.

[18] Niranjan Hasabnis, Ashish Misra, and R Sekar. 2012. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization.* 135–144.

[19] Reed Hastings and Bob Joyce. 1992. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In *Proceedings of the Winter 1992 USENIX Conference.* 125–138.

[20] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: a safe dialect of C.. In *USENIX Annual Technical Conference, General Track.* 275–288.

[21] Nick P. Johnson, Hanjun Kim, Prakash Prabhu, Ayal Zaks, and David I. August. 2012. Speculative Separation for Privatization and Reductions. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 359–370. https://doi.org/10.1145/2254064.2254107

[22] Tina Jung, Fabian Ritter, and Sebastian Hack. 2021. PICO: a Presburger in-bounds check optimization for compiler-based memory safety instrumentations. *ACM Transactions on Architecture and Code Optimization (TACO)* 18, 4 (2021), 1–27.

[23] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[24] Sanghoon Lee and James Tuck. 2013. Automatic Parallelization of Fine-Grained Metafunctions on a Chip Multiprocessor. *ACM Trans. Archit. Code Optim.* 10, 4, Article 30 (Dec. 2013), 26 pages. https://doi.org/10.1145/2541228.2541237

[25] Daiping Liu, Mingwei Zhang, and Haining Wang. 2018. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.* 1635–1648.

[26] Jiuyue Ma, Xiufeng Sui, Ninghui Sun, Yupeng Li, Zihao Yu, Bowen Huang, Tianni Xu, Zhicheng Yao, Yun Chen, Haibin Wang, et al. 2015. Supporting differentiated services in computers via programmable architecture for resourcing-on-demand (PARD). In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems.* 131–143.

[27] Angelo Matni, Enrico Armenio Deiana, Yian Su, Lukas Gross, Souradip Ghosh, Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Ishita Chaturvedi, David I. August, and Simone Campanoni. 2022. NOELLE Offers Empowering LLVM Extensions. In *International Symposium on Code Generation and Optimization, 2022. CGO 2022.*

[28] Matt Miller. 2019. MSRC security report. https://github.com/microsoft/MSRC-Security-Research/tree/master/presentations/2019_02_BlueHatIL. Accessed: 2022-3-1.

[29] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. 2018. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18).* 919–936.

[30] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. 2012. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *2012 39th Annual International Symposium on Computer Architecture (ISCA).* IEEE, 189–200.

[31] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. 2015. Everything you want to know about pointer-based checking. In *1st Summit on Advances in Programming Languages (SNAPL 2015).* Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[32] Myoung Jin Nam, Periklis Akritidis, and David J Greaves. 2019. FRAMER: a tagged-pointer capability system with memory safety applications. In *Proceedings of the 35th Annual Computer Security Applications Conference.* 612–626.

[33] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. 2008. Parallelizing Security Checks on Commodity Hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, USA) *(ASPLOS XIII)*. Association for Computing Machinery, New York, NY, USA, 308–318. https://doi.org/10.1145/1346281.1346321

[34] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2018. Intel mpx explained: A cross-layer analysis of the intel mpx system stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2, 2 (2018), 1–30.

[35] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I August. 2005. Automatic thread extraction with decoupled software pipelining. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05).* IEEE, 12–pp.

[36] Harish Patil and Charles Fischer. 1997. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software: Practice and Experience* 27, 1 (1997), 87–110.

[37] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2010. Combined iterative and model-driven optimization in an automatic parallelization framework. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE, 1–11.

[38] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J Bridges, and David I August. 2008. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization.* 114–123.

[39] Bin Ren, Sriram Krishnamoorthy, Kunal Agrawal, and Milind Kulkarni. 2017. Exploiting vector and multicore parallelism for recursive, data-and task-parallel programs. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* 117–130.

[40] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12).* 309–318.

[41] Yulei Sui, Ding Ye, Yu Su, and Jingling Xue. 2016. Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions. *IEEE Transactions on Reliability* 65, 4 (2016), 1682–1699.

[42] M. Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, Ryan Piersma, and Simha Sethumadhavan. 2021. No-FAT: Architectural Support for Low Overhead Memory Safety Checks. In *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA-48).* Worldwide Event, 916–929.

[43] Evangelos Vlachos, Michelle L Goodstein, Michael A Kozuch, Shimin Chen, Babak Falsafi, Phillip B Gibbons, and Todd C Mowry. 2010. ParaLog: Enabling and accelerating online parallel monitoring of multithreaded applications. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems.* 271–284.

[44] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. 2015. High system-code security with low overhead. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 866–879.

[45] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G Neumann, Simon W Moore, Robert NM Watson, et al. 2019. Cherivoke: Characterising pointer revocation using cheri capabilities for temporal memory safety. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 545–557.

[46] Hongfa Xue, Yurong Chen, Fan Yao, Yongbo Li, Tian Lan, and Guru Venkataramani. 2017. Simber: Eliminating redundant memory bound checks via statistical inference. In *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 413–426.

[47] Suan Hsi Yong and Susan Horwitz. 2005. Using static analysis to reduce dynamic analysis overhead. *Formal Methods in System Design* 27, 3 (2005), 313–334.

[48] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. 2021. SANRAZOR: Reducing Redundant Sanitizer Checks in C/C++ Programs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 479–494. https://www.usenix.org/conference/osdi21/presentation/zhang

Received 2023-02-16; accepted 2023-05-03