June 17, 2023
Orlando, FL, USA

Association for Computing Machinery

Advancing Computing as a Science & Profession

SOAP '23

Proceedings of the 12th ACM SIGPLAN International Workshop on

the State Of the Art in Program Analysis

Edited by:
Pietro Ferrara and Liana Hadarean

Sponsored by:
ACM SIGPLAN

Co-located with:
PLDI '23

Association for Computing Machinery, Inc.
1601 Broadway, 10th Floor
New York, NY 10019-7434
USA

ACM ISBN: 979-8-4007-0170-2

Cover photo
Title: "Lake Cane At Sunrise Orlando Florida Landscape Photography"
Photographer: Giuseppe Milo
License: Creative Commons Attribution 3.0 Unported https://creativecommons.org/licenses/by/3.0/deed.en
Cropped from original: https://upload.wikimedia.org/wikipedia/commons/8/8c/Lake_Cane_At_Sunrise_Orlando_Florida_Landscape_Photography_%28190809939%29.jpeg

Production: Conference Publishing Consulting
D-94034 Passau, Germany, info@conference-publishing.com

Welcome from the Chairs

The 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '23) is co-located with the 44th ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '23). In line with past workshops, SOAP '23 aims to bring together the members of the program analysis community to share new developments and shape innovations in program analysis. This edition of SOAP had eleven submissions, each reviewed by three reviewers. Seven were accepted (64% acceptance rate), and three others were conditionally accepted and then shepherd into final acceptance.

Along with the accepted papers, SOAP '23 features three invited talks by leading members of the program analysis community: Tucker Taft (Adacore), Ranjit Jhala (UCSD) and William Hallahan (Binghamton University).

We would like to commend the efforts of the six members of the program committee, who donated their valuable time and effort to make the reviewing process possible. We also thank the PLDI chairs, the workshops chairs and the ACM staff for their continued support in making this workshop possible. We hope you enjoy the talks at SOAP'23 and look forward to enlightening discussions.

June 2023

Pietro Ferrara
Liana Hadarean

Committees Program Chairs:

Pietro Ferrara (Università Ca' Foscari, Italy)
Liana Hadarean (Amazon, USA)

Program Committee:

Vincenzo Arceri (University of Parma, Italy)
Fraser Brown (CMU, USA)
Cezara Drăgoi (AWS / ENS, France)
Marco Eilers (ETH Zurich, Switzerland)
Daniel Kaestner (AbsInt, Germany)
Raphaël Monat (Inria and University of Lille, France)

Steering Committee:

Laure Gonnord (Grenoble-INP/LCIS, France)
Neville Grech (University of Malta, Malta)
Ben Hermann (Technische Universität Dortmund, Germany)
Padmanabhan Krishnan (Oracle Labs, Australia)
Thierry Lavoie (Synopsys, Canada)
Lisa Nguyen Quang Do (Google, Switzerland)
Christoph Reichenbach (Lund University, Sweden)
Laura Titolo (NIA/NASA LaRC, USA)
Omer Tripp (Amazon, USA)
Caterina Urban (INRIA and École Normale Supérieure, France)

Contents

Frontmatter

Papers

Combining E-Graphs with Abstract Interpretation

Samuel Coward*
s.coward21@imperial.ac.uk
Electrical and Electronic Engineering
Imperial College London
UK

George A. Constantinides
g.constantinides@imperial.ac.uk
Electrical and Electronic Engineering
Imperial College London
UK

Theo Drane
theo.drane@intel.com
Numerical Hardware Group
Intel Corporation
USA

Abstract

E-graphs are a data structure that compactly represents equivalent expressions. They are constructed via the repeated application of rewrite rules. Often in practical applications, *conditional* rewrite rules are crucial, but their application requires the detection – at the time the e-graph is being built – that a condition is valid in the domain of application. Detecting condition validity amounts to proving a property of the program. Abstract interpretation is a general method to learn such properties, traditionally used in static analysis tools. We demonstrate that abstract interpretation and e-graph analysis naturally reinforce each other through a tight integration because (i) the e-graph clustering of equivalent expressions induces natural precision refinement of abstractions and (ii) precise abstractions allow the application of deeper rewrite rules (and hence potentially even greater precision). We develop the theory behind this intuition and present an exemplar interval arithmetic implementation, which we apply to the FPBench suite.

CCS Concepts: • **Theory of computation → Abstraction**; **Equational logic and rewriting**; • **Mathematics of computing → Interval arithmetic**.

Keywords: abstract interpretation, interval arithmetic, static analysis, e-graph

ACM Reference Format:
Samuel Coward, George A. Constantinides, and Theo Drane. 2023. Combining E-Graphs with Abstract Interpretation. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '23), June 17, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3589250.3596144

*Also with Numerical Hardware Group, Intel Corporation.

SOAP '23, June 17, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0170-2/23/06…$15.00
https://doi.org/10.1145/3589250.3596144

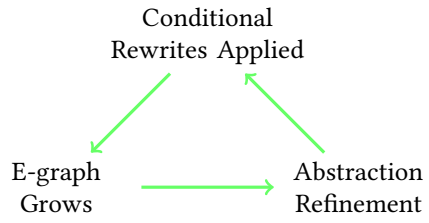


Figure 1. The positive feedback loop between e-graph exploration and abstraction refinement.

1 Introduction

Equivalence graphs, commonly called e-graphs, provide a compact representation of equivalence classes (e-classes) of expressions, where the notion of equivalence is with respect to some concrete semantics [24]. The recent egg tool introduced *e-class analysis*, a technique to enable program analysis over an e-graph, attaching analysis data to each e-class [30]. This paper formalises some of the concepts required to produce e-class analyses enabling e-graph growth via conditional rewrites. We show that partitioning expressions into e-classes gives rise to a natural lattice-theoretic interpretation for abstract interpretation (AI), resulting in the generation of precise abstractions. By interleaving conditional e-graph rewriting and program analysis the exploration power of the e-graph can be greatly enhanced. Figure 1 visualizes this positive feedback loop. Figure 2 provides an example, in which interval analyses of equivalent expressions are combined to produce tighter enclosing intervals.

We develop the general theoretical underpinnings of AI on e-graphs, exploiting rewrites to produce tight abstractions using a lattice-theoretic formalism. We also provide a sound interpretation of cycles naturally arising in e-graphs, corresponding to extracting abstract fixpoint equations, and show that a known interval algorithm, the Krawczyk method, results as a special case.

The ideas presented here inspired the egg developers to implement interval analysis as part of their PLDI 2022 tutorial[1]. This paper extends and generalizes an abstract presented at the EGRAPHS workshop [11] in the following ways:

- formalization of AI with e-graphs in lattice theory,
- relating fixpoints to e-graph cycles to automatically discover iterative abstract refinement methods,

[1]https://github.com/egraphs-good/egg-tutorial-pldi-2022

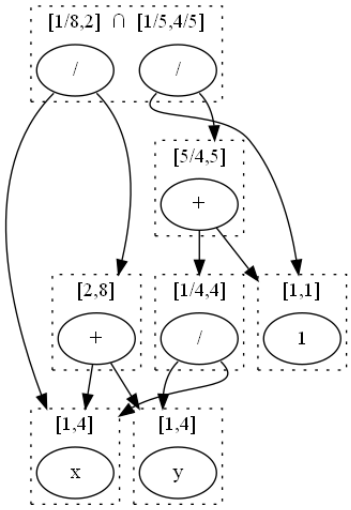


Figure 2. E-graph containing equivalent real arithmetic expressions $\frac{x}{x+y}$ and $\frac{1}{1+(y/x)}$ in the root e-class (at the top). Intervals are associated with each e-class. Input constraints are propagated upwards via an e-class analysis.

- techniques to capture relationships between variables,
- an interval arithmetic implementation with associated expression bounding results.

A short background overview is provided in §2. In §3 we present the theoretical application of AI to e-graphs and then demonstrate its viability using an interval arithmetic implementation in §4. Lastly, in §5 we present results.

2 Background

The e-graph data structure is commonly found in theorem provers and solvers [16, 17]. It represents multiple expressions as a graph (e.g. Figure 2), where the nodes represent functions, grouped together into collections of e-classes. Edges connect nodes to e-classes, as a given sub-expression may be implemented using any of the nodes found in the child e-class. E-graphs are often combined with an optimisation technique called equality saturation [21, 29, 30], which deploys equivalence preserving transformations to monotonically grow the e-graph and discover alternative equivalent expressions. A recent resurgence of e-graph research [30] has seen the technique applied to floating point numerical stability analysis [25], mapping programs onto hardware accelerators [26] and optimizing hardware designs [12]. Some works make heavy use of conditional rewrites of the form $\phi \Rightarrow \ell \rightarrow r$, for example $x \neq 0 \Rightarrow x/x \rightarrow 1$, which require the e-graph construction algorithm to determine whether the rewrite is applicable in each case [12, 30].

AI has primarily been applied to static program analysis [8], where computer programs are automatically analyzed without actually being executed. It uses the theory of abstraction to consider over-approximations of the program behaviour using alternative interpretations [6, 8]. Existing tools have incorporated term rewriting to refine their abstractions [15]. In §3.4 we will discuss relationships between variables allowing the framework to infer the consequences of conditional branches. Previous work on constraint programming has combined constraint awareness with interval analysis [28], whilst Granger identified that constraints are typically better evaluated in the abstract domain [19].

For representing disjunctions of intervals, researchers have developed Linear Decision Diagrams (LDDs) and Range Decision Diagrams (RDDs) which are extensions of Binary Decision Diagrams (BDDs) [18, 20]. Previous work on abstract congruence closure [2, 3] has exploited e-graphs to combine different abstract domains [5], but we are not aware of existing work exploiting the tight interaction between AI and e-graph construction.

3 Theory

3.1 Abstraction

From a theoretical viewpoint, AI [7] is concerned with relationships between lattices, defined via Galois connections.

Definition 1 (Lattice)**.** A lattice is a partially ordered set (poset) $\langle L, \leq \rangle$, such that $\forall a, b \in L$ the least upper bound (join) $a \sqcup b$ and the greatest lower bound (meet) $a \sqcap b$ both exist.

Definition 2 (Galois connection)**.** Given a poset $\langle C, \sqsubseteq \rangle$, corresponding to the concrete domain, and a poset $\langle \mathcal{A}, \preccurlyeq \rangle$, corresponding to the abstract domain, then a function pair $\alpha \in C \rightarrow \mathcal{A}, \gamma \in \mathcal{A} \rightarrow C$, defines a Galois connection iff

$$\forall P \in C.\ \forall \overline{P} \in \mathcal{A}.\ \alpha(P) \preccurlyeq \overline{P} \Leftrightarrow P \sqsubseteq \gamma(\overline{P}),$$

$$\text{written } \langle C, \sqsubseteq \rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle \mathcal{A}, \preccurlyeq \rangle.$$

The pair (α, γ) define the abstraction and concretization respectively, allowing us to over-approximate (i.e. abstract) concrete properties in C with abstract properties in $\mathcal{A}$.

Definition 3 (Sound abstraction [7])**.** $\overline{P} \in \mathcal{A}$ is a sound abstraction of a concrete property $P \in C$ iff $P \sqsubseteq \gamma(\overline{P})$.

Consider expressions evaluated over a domain $\mathcal{D}$. By imposing a canonical ordering on the variable set, we work within a defined subset $I \subseteq \mathcal{D}^n$, which encodes a precondition on the set of (input) variable values. Now consider a (concrete) semantics of expressions $[\![\cdot]\!]. \in \text{Expr} \rightarrow I \rightarrow \mathcal{D}$, where Expr denotes the set of expressions, so $[\![e]\!]_\rho$ denotes the interpretation of expression e under execution environment (assignment of variables to values) $\rho \in I$. Let $[\![e]\!] = \{[\![e]\!]_\rho \mid \rho \in I\}$. The e-graph data structure encodes equivalence under concrete semantics, which we shall now define precisely.

Definition 4 (Congruence)**.** Two expressions e_a and e_b are congruent, $e_a \cong e_b$, iff $[\![e_a]\!]_\rho = [\![e_b]\!]_\rho$ for all $\rho \in I$.

Lemma 1. If $e_a \cong e_b$ and $\overline{P}$ is a sound abstraction of $[\![e_a]\!]$, then $\overline{P}$ is a sound abstraction of $[\![e_b]\!]$.

Proof. by definition of congruence. □

This lemma implies that a sound abstraction of one expression in an e-class is a sound abstraction of all expressions in the e-class. Precision refinement relies on the following, which is a specialization of the more general result [10].

Lemma 2. For any two sound abstractions $\overline{P}_a$ and $\overline{P}_b$ of P, the meet $\overline{P}_a \sqcap \overline{P}_b$ is also a sound abstraction of P.

Proof. $P \sqsubseteq \gamma(\overline{P}_a)$ (sound abstraction) $\Rightarrow \alpha(P) \preccurlyeq \overline{P}_a$ (Galois connection) and similarly $P \sqsubseteq \gamma(\overline{P}_b) \Rightarrow \alpha(P) \preccurlyeq \overline{P}_b$. Therefore $\alpha(P) \preccurlyeq \overline{P}_a \sqcap \overline{P}_b$ (meet definition) and hence $P \sqsubseteq \gamma(\overline{P}_a \sqcap \overline{P}_b)$ (Galois connection). □

3.2 Application to E-graphs

Consider an e-graph. Let S denote the set of e-classes, and $\mathcal{N}_s$ the set of nodes in the equivalence class $s \in S$. With each e-class, associate an abstraction $A \in \mathcal{A}$ and write $\mathcal{A}[\![s]\!] = A$. Interpreting a k-arity node n of function f with children classes $s_1, ..., s_k$, using an arbitrary sound abstraction $\bar{f}$:

$$\mathcal{A}[\![n]\!] = \bar{f}(\mathcal{A}[\![s_1]\!], ..., \mathcal{A}[\![s_k]\!]). \tag{1}$$

0-arity nodes are either constants with exact abstractions in $\mathcal{A}$ or variables with user specified abstract constraints.

For acyclic e-graphs, we propagate the known abstractions upwards using (1), taking the greatest lower bound (meet) across all nodes in the e-class.

$$\mathcal{A}[\![s]\!] = \bigsqcap_{n \in \mathcal{N}_s} \mathcal{A}[\![n]\!] \tag{2}$$

The propagation algorithm is described in Figure 3, where

$$\texttt{make}(n) = \mathcal{A}[\![n]\!] \text{ and } \texttt{meet}(A_1, A_2) = A_1 \sqcap A_2.$$

These functions are analogous to those described for an e-class analysis [30], but replace their `join` with a `meet`.

By lifting the abstract analysis from expressions to e-classes of expressions we construct a more precise analysis. In the abstract domain the notion of equivalence is different, $n_a, n_b \in \mathcal{N}_s \not\Rightarrow \mathcal{A}[\![n_a]\!] = \mathcal{A}[\![n_b]\!]$, which results in tighter abstractions since the meet corresponds to a more precise abstraction in $\mathcal{A}$. In the algorithm in Figure 3, by initializing the `workqueue` with only the modified e-classes after application of a rewrite, the abstract properties of the e-graph can be evaluated on the fly. For an acyclic e-graph, the propagation of each merge is worst case linear in the size of the e-graph. On-the-fly evaluation facilitates conditional rewrite application as more precise properties are discovered during construction. This is used in §4.

A positive feedback loop is created by combining AI and e-graphs (Figure 1). A larger space of equivalent expressions is explored as more rewrites can be proven to be valid at exploration time. In turn, expression abstractions are further refined by discovering more equivalent expressions, allowing even more valid rewrites, and the cycle continues. Additionally, several equivalent expressions may contribute to the tight final abstraction. An example is shown in §4.

```
workqueue = egraph.classes().leaves()
while !workqueue.is_empty()
  s = workqueue.dequeue()
  for n in s.nodes()
    skip_node = false
    for child_s in n.children()
      if child_s.uninitialized
        workqueue.enqueue(s)
        skip_node = true
    if skip_node
        continue
    elif s.uninitialized
        s.data = make(n)
        s.uninitialized = false
        workqueue.enqueue(s.parents())
    elif !(s.data<=meet(s.data,make(n)))
        s.data = meet(s.data,make(n))
        workqueue.enqueue(s.parents())
```

Figure 3. Pseudocode for abstract property propagation in an e-graph.

3.3 Cyclic E-graphs and Fixpoints

Cyclic e-graphs arise when an expression is equivalent to a sub-expression of itself with respect to concrete semantics, for example $e \times 1 \cong e$. Let $e \cong e'$ where e appears as a subterm in e'. Treating other subterms as absorbed into the function, let $f : \mathcal{D} \to \mathcal{D}$ be the interpretation of e' as a (concrete) function of $[\![e]\!]_\rho$, so that – in particular – $f([\![e]\!]_\rho) = [\![e]\!]_\rho$ due to the congruence and hence $[\![e]\!] = \{f([\![e]\!]_\rho) \mid \rho \in I\}$. Abstracting f via a sound abstraction $\bar{f}$, yields the corresponding abstract fixpoint equation $a = a \sqcap \bar{f}(a)$ where the meet operation arises from (2).

Now consider the function $\tilde{f}(a) = a \sqcap \bar{f}(a)$. The decreasing sequence defined by $a_{n+1} = \tilde{f}(a_n)$ corresponds to applying the abstract property propagation around a cycle in the e-graph, given an initial sound abstraction a_0 of $[\![e]\!]$.

Lemma 3. $\alpha([\![e]\!])$ is a fixpoint of $\tilde{f}$.

Proof.

$$\begin{aligned} \alpha([\![e]\!]) &= \alpha\left(\{f([\![e]\!]_\rho) \mid \rho \in I\}\right) && \text{(congruence)} \\ &\preccurlyeq \bar{f}(\alpha([\![e]\!])) && \text{(sound abstraction)} \end{aligned}$$

Hence $\tilde{f}(\alpha([\![e]\!])) = \alpha([\![e]\!]) \sqcap \bar{f}(\alpha([\![e]\!])) = \alpha([\![e]\!])$ (meet definition). □

Lemma 4. a_n is a sound abstraction of $[\![e]\!]$ for all $n \in \mathbb{N}$.

Proof. By induction, $[\![e]\!] \sqsubseteq \gamma(a_0)$ and assume $[\![e]\!] \sqsubseteq \gamma(a_n)$. $[\![e]\!] = \{f([\![e]\!]_\rho) \mid \rho \in I\} \sqsubseteq \gamma(\bar{f}(a_n))$ (sound abstraction of f). Hence $a_{n+1} = a_n \sqcap \bar{f}(a_n)$ is a sound abstraction of $[\![e]\!]$ (Lemma 2) for all n. □

Collecting these results, for some fixpoint a^* we have

$$\alpha([\![e]\!]) \preccurlyeq a^* \preccurlyeq \ldots \preccurlyeq a_1 \preccurlyeq a_0. \tag{3}$$

Thus computing abstractions around the loop refines the abstraction and is guaranteed to terminate if the lattice $\langle \mathcal{A}, \preccurlyeq \rangle$ satisfies the descending chain condition, as any finite abstract domain will [4]. Note that the fixpoint a^* is neither greatest, least nor unique. This can be seen because the top and bottom elements of the lattice are both also fixedpoints of $\tilde{f}$, but from (3) the computed fixedpoint is between these two other fixedpoints. The algorithm in Figure 3 will correctly apply abstract property propagation around loops, terminating if the sequence a_n converges in a finite number of steps. Of course for abstract domains with infinite descending chains standard techniques such as narrowing apply [9].

3.4 Relational Domains

For simplicity of exposition, we focus on non-relational domains in the discussion above. However, it is important to note that the combination of non-relational domains with the *relational* information provided by rewrite rules provides a stronger analysis than classical non-relational domains. By introducing an additional ASSUME node that captures the domain refinement implied by design constraints, relationships between variables can be captured, enabling further rewrites. We illustrate this point via an example.

$$x == y \,?\, x + y : 0. \tag{4}$$

A human quickly realises the relationship between x and y in the true branch and can replace $x + y$ by $2 \times y$. In the e-graph framework, the expression is automatically rewritten to identify 'new' terms which simplify the conditional reasoning. As the e-graph grows (4) will be found to be equivalent to:

$$z == 0 \,?\, z + 2 \times y : 0, \quad \text{where } z = x - y. \tag{5}$$

The e-graph identifies a new variable, z, which must be a constant, zero, on the true branch. The ASSUME node is introduced by rewriting the ternary operation.

$$z == 0 \,?\, \texttt{ASSUME}(z + 2 \times y,\, z == 0) : \texttt{ASSUME}(0,\, z \neq 0) \tag{6}$$

The $z == 0$ constraint is automatically pushed down the expression tree. At the point at which the propagated condition reaches z it is automatically rewritten to constant zero. The e-graph continues to grow, until it discovers the simplified equivalent expression, where the ASSUME nodes are dropped.

$$z == 0 \,?\, 2 \times y : 0, \quad \text{where } z = x - y. \tag{7}$$

We combined an existing hardware optimization e-graph tool [12] with the program analysis techniques presented here, where we introduce ASSUME nodes in more detail [13]. In this work, we implement rewriting over real valued expressions but in [13] we apply the techniques presented here to fixed width integer valued expressions.

4 Implementation

To demonstrate the theory described above, we implement interval arithmetic (IA) [23] for real valued expressions using the extensible egg library, as an e-class analysis [30]. We consider a concrete domain corresponding to sets of extended real numbers, i.e. $\mathcal{C} = \mathcal{P}(\mathbb{R} \cup \{-\infty, +\infty\})$ where $\mathcal{P}$ denotes the power set. We associate each expression with a binary64 [1] valued interval (a finite abstract domain),

$$\mathcal{A} = \{[a, b] \mid a \leq b, a, b \in \texttt{binary64} \setminus \{\texttt{NaN}\}\} \cup \{\emptyset\}.$$

In this setting the abstraction and concretization functions are as follows (infima/suprema always exist in this setting):

$$\alpha(X) = [\text{round\_down}(\inf X), \text{round\_up}(\sup X)] \tag{8}$$

$$\gamma([a, b]) = [a, b] \tag{9}$$

$$\alpha(\emptyset) = \emptyset, \gamma(\emptyset) = \emptyset \tag{10}$$

For this work we supported the following set of operators, +, -, ×, /, $\sqrt{}$, pow, exp and ln. To ensure correctness, we use 'outwardly rounded IA' which conservatively rounds upper bounds towards $+\infty$ (round_up) and lower bounds towards $-\infty$ (round_down) [22, 23]. For the elementary functions, $\sqrt{}$, exp and ln, we use default library implementations but are unable to control the rounding mode, so conservatively add or subtract one unit in the last place for upper and lower bounds respectively. Provided NaNs do not appear in the input expression evaluation they are not generated by the e-graph exploration. Furthermore abstract intervals containing -0 shall be mapped by γ to sets containing $0 \in \mathcal{C}$.

In this case we use an abstraction of a given function f, $\bar{f} = \alpha \circ f \circ \gamma$. For e-class s under this interpretation, (2) uses the intersection operation, the meet operation of the lattice of intervals.

$$\mathcal{A}[\![s]\!] = \bigcap_{n \in N_s} [\![n]\!] \tag{11}$$

This relationship generates monotonically narrowing interval abstractions. 0-arity nodes represent either constants associated with degenerate intervals or variables taking user defined interval constraints.

The classical problem of interval arithmetic is the so-called 'dependency problem', arising because the domain does not capture correlations between multiple occurrences of a single variable. Consider $x \in [0, 1]$, under classical IA:

$$\mathcal{A}[\![x - x]\!] = [0, 1] - [0, 1] = [0 - 1, 1 - 0] = [-1, 1]. \tag{12}$$

The e-graph framework discovers, via term rewriting, $x - x \cong 0$ and by (11) the expression is now correctly abstracted by the (much tighter) degenerate interval $[0, 0]$.

Table 1. Additional IA optimization rewrites.

Class	Rewrite	Condition
Factor	$ab \pm ac \longrightarrow a(b \pm c)$	True
Binom.	$1/(1-a) \longrightarrow 1 + a/(1-a)$	$0 \notin [\![1-a]\!]$
Frac	$b/c \pm a \longrightarrow (b \pm ac)/c$	$0 \notin [\![c]\!]$
Div. [23]	$a/b \longrightarrow 1/(b/a)$ $a/b \longrightarrow 1 + (a-b)/b$	$0 \notin [\![a]\!] \cup [\![b]\!]$ $0 \notin [\![b]\!]$
Poly	$a^2 - 1 \longrightarrow (a-1)(a+1)$	True
Elem.	$\ln(e^a) \longrightarrow a$	True

We use a set of 39 rewrites, defining equivalences of real valued expressions. The basic arithmetic rewrites are commutativity, associativity, distributivity, cancellation and idempotent operation reduction across addition, subtraction, multiplication and division. Conversion rewrites describe the natural equivalence between the power function and multiplication/division. Table 1 contains the remaining rewrites.

Conditional rewrites, e.g. "Div.", are only valid on a subset of the input domain. Via an IA the e-graph can prove validity of such rules. In (13) IA can confirm that $0 \notin [\![x + y]\!]$, in order to remove multiple occurrences of variables resulting in expression bound improvements.

$$\frac{x+y}{x+y+1} \longrightarrow \ldots \longrightarrow \frac{1}{1+\frac{1}{x+y}} \tag{13}$$

Multiple expressions in an e-class can independently contribute to a tight abstraction. Consider the following equivalent expressions for variables $x \in [0,1]$ and $y \in [1,2]$.

$$1 - \frac{2y}{x+y} \in \left[-3, \frac{1}{3}\right] \tag{14}$$

$$\cong \frac{x-y}{x+y} \in [-2, 0] \tag{15}$$

$$\cong \frac{2x}{x+y} - 1 \in [-1, 1]. \tag{16}$$

All three reside in the same e-class within an e-graph with associated interval $[-3, \frac{1}{3}] \cap [-2, 0] \cap [-1, 1] = [-1, 0]$. Thus, given the first expression (14), the e-graph generates a tight interval enclosure using two distinct equivalent expressions for the upper (15) and lower (16) bounds.

5 Results

Based on the theory introduced above we describe a real valued expression bounding tool in Rust using the egg library. All test cases were run on an Intel i7-10610U CPU.

5.1 Benchmarks

We evaluate the implementation using 40 benchmarks from the FPTaylor [27] supported subset of the FPBench benchmark suite [14]. We allow four iterations of e-graph rewriting, as further rewriting iterations do not yield significant improvements in interval width on these modest benchmarks.

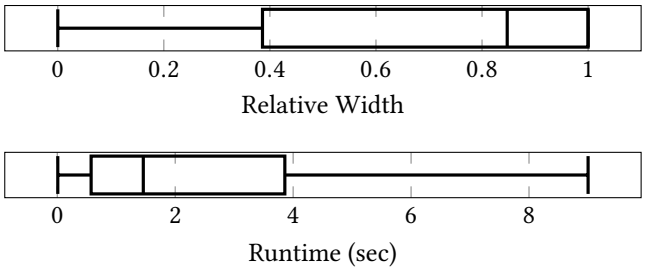


Figure 4. Relative interval width (optimized width/naive width) and runtime boxplots to demonstrate the distribution of results on the FPBench suite.

Across these benchmarks, the inclusion of IA and domain specific rewrites increased the number of e-graph nodes by 4% on average but by up to 84%. This demonstrates the additional rewrites that have been applied as a result of combining e-graphs and AI. Figure 4 summarises the distribution of the interval width improvement and runtime across the benchmarks. The relative width boxplot shows that, on average, e-graph rewriting reduced the interval to 85% of the width of the naive interval approximation. In a number of cases, the interval obtained from e-graph rewriting was reduced by almost 100%. The runtime boxplot shows that runtimes remained small. There is little correlation between the runtime and bound improvement. The overhead of incorporating IA into the e-graph increased runtimes by less than 1% on average.

5.2 Iterative Method Discovery

In §3.3 we discussed how cyclic e-graphs can discover iterative refinements. The Krawczyk method [23] is a known algorithm to generate increasingly precise element-wise interval enclosures of solutions of linear systems of equations $Ax = b$, where A is an n-by-n matrix and b is an n-dimensional vector. Letting X^0 be an initial interval enclosure of the solutions, the Krawcyzk method uses an update formula of the form,

$$X^{k+1} = \left(Yb + (I - YA)X^k\right) \cap X^k, \text{ where } Y = \text{mid}(A)^{-1}.$$

mid(A) is the element-wise interval midpoint of the matrix. This sequence, via interval extension and intersection, corresponds to a sequence of tightening bounds on the solution x, which converges provided the matrix norm $||I - YA|| < 1$.

We consider a specific instance of this problem,

$$\begin{pmatrix} 1 & y \\ y & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}, \text{ where } y \in \left[-\frac{1}{2}, \frac{1}{2}\right]. \tag{17}$$

$$X_1^{k+1} = \left(b_1 - yX_2^k\right) \cap X_1^k, \; X_2^{k+1} = \left(b_2 - yX_1^k\right) \cap X_2^k. \tag{18}$$

A naive solution in the concrete domain, $x = A^{-1}b$, yields,

$$x_1 = \frac{1}{1-y^2}(b_1 - b_2 y), \; x_2 = \frac{1}{1-y^2}(b_2 - b_1 y). \tag{19}$$

Initialising the e-graph with these expressions, the solution for x_1 can be automatically rewritten such that (18) arises in the abstract domain. The "Binom." rewrite from Table 1 introduces a loop into the e-graph, which when combined with distributivity rules and "Factor" from Table 1 yields,

$$x_1 = b_1 - y\left(b_2 + (b_2y^2 - b_1y)\frac{1}{1-y^2}\right). \tag{20}$$

After applying, "Frac", and cancelling, the e-graph contains

$$x_1 = b_1 - y(b_2 - b_1y)\frac{1}{1-y^2} = b_1 - yx_2. \tag{21}$$

When a cycle is introduced, the IA update procedure will continue to iteratively evaluate the loop, taking the intersection with the previous iteration as described in §3.3.

6 Conclusion

We present a combination of abstract interpretation and e-graphs, demonstrating the natural interpretation of e-class partitions as meet operators in a lattice, resulting in precise abstractions. Of key importance is the positive feedback loop between e-graph exploration and abstraction refinement, as the precision then allows the application of conditional rewrite rules, which can be applied in many domains and may further improve abstraction precision. An exemplar interval arithmetic implementation has demonstrated the value of this idea, including automated discovery of a known algorithm for iterative refinement. Furthermore, an existing hardware optimization e-graph application has already benefitted from incorporating these analysis techniques. We believe that other e-graph applications could also incorporate AI to extend their capabilities for relatively low overhead. Future work will explore additional abstract domains and their incorporation into e-graph optimization tools.

References

[1] 2008. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008* (2008), 1–70.

[2] Leo Bachmair and Ashish Tiwari. 2000. Abstract congruence closure and specializations. In *Lecture Notes in Artificial Intelligence (Subseries of Lecture Notes in Computer Science)*, David McAllester (Ed.), Vol. 1831. Springer. https://doi.org/10.1007/10721959_5

[3] Leo Bachmair, Ashish Tiwari, and Laurent Vigneron. 2003. Abstract congruence closure. *Journal of Automated Reasoning* 31, 2 (2003). https://doi.org/10.1023/B:JARS.0000009518.26415.49

[4] Garrett Birkhoff. 1958. Von Neumann and Lattice Theory. *Bull. Amer. Math. Soc.* 64, 3 (1958). https://doi.org/10.1090/S0002-9904-1958-10192-5

[5] Bor Yuh Evan Chang and K. Rustan M. Leino. 2005. Abstract interpretation with alien expressions and heap structures. In *Lecture Notes in Computer Science*, Vol. 3385. Springer. https://doi.org/10.1007/978-3-540-30579-8_11

[6] Patrick Cousot. 2001. Abstract Interpretation Based Formal Methods and Future Challenges. In *Informatics: 10 Years Back, 10 Years Ahead.* Springer. https://doi.org/10.1007/3-540-44577-3_10

[7] Patrick Cousot. 2021. *Principles of Abstract Interpretation* (1st ed.). MIT Press.

[8] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, Vol. Part F130756. Association for Computing Machinery. https://doi.org/10.1145/512950.512973

[9] Patrick Cousot and Radhia Cousot. 1992. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 631 LNCS. Springer. https://doi.org/10.1007/3-540-55844-6_142

[10] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. 2011. The reduced product of abstract domains and the combination of decision procedures. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 6604 LNCS. Springer. https://doi.org/10.1007/978-3-642-19805-2_31

[11] Samuel Coward, George A. Constantinides, and Theo Drane. 2022. Abstract Interpretation on E-Graphs. https://arxiv.org/abs/2203.09191

[12] Samuel Coward, George A. Constantinides, and Theo Drane. 2022. Automatic Datapath Optimization using E-Graphs. In *IEEE 29th Symposium on Computer Arithmetic (ARITH)*. IEEE, 43–50. https://doi.org/10.1109/ARITH54963.2022.00016

[13] Samuel Coward, George A. Constantinides, and Theo Drane. 2023. Automating Constraint-Aware Datapath Optimization using E-Graphs. (2023). https://arxiv.org/abs/2303.01839

[14] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Chen Qiu, Alexander Sanchez-Stern, and Zachary Tatlock. 2017. Toward a standard benchmark format and suite for floating-point analysis. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 10152 LNCS. Springer. https://doi.org/10.1007/978-3-319-54292-8_6

[15] Marc Daumas and Guillaume Melquiond. 2010. Certification of bounds on expressions involving rounded operators. *ACM Trans. Math. Software* 37, 1 (2010). https://doi.org/10.1145/1644001.1644003

[16] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT Solver. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 4963 LNCS. Springer. https://doi.org/10.1007/978-3-540-78800-3_24

[17] David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: A theorem prover for program checking. *J. ACM* 52, 3 (2005). https://doi.org/10.1145/1066100.1066102

[18] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2021. Disjunctive Interval Analysis. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 12913 LNCS. Springer International Publishing, Cham, 144–165. https://doi.org/10.1007/978-3-030-88806-0_7

[19] Philippe Granger. 1992. Improving the results of static analyses of programs by local decreasing iterations. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 652 LNCS. Springer. https://doi.org/10.1007/3-540-56287-7_95

[20] Arie Gurfinkel and Sagar Chaki. 2010. Boxes: A symbolic abstract domain of boxes. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 6337 LNCS. Springer Berlin Heidelberg, Berlin, Heidelberg, 287–303. https://doi.org/10.1007/978-3-642-15769-1_18

[21] Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: A goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery.

[22] Ulrich Kulisch. 1981. *Computer Arithmetic in Theory and Practice*. Academic press. https://doi.org/10.1016/c2013-0-11018-5

[23] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. 2009. *Introduction to Interval Analysis*. SIAM. https://doi.org/10.1137/1.9780898717716

[24] Charles Gregory Nelson. 1980. *Techniques for program verification*. Ph. D. Dissertation. Stanford University.

[25] Pavel Panchekha, Alex Sanchez-Stern, James R Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. *ACM SIGPLAN Notices* 50, 6 (2015), 1–11.

[26] Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock. 2021. Pure tensor program rewriting via access patterns (representation pearl). In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*. Association for Computing Machinery. https://doi.org/10.1145/3460945.3464953

[27] Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. 2018. Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions. *ACM Transactions on Programming Languages and Systems* 41, 1 (2018). https://doi.org/10.1145/3230733

[28] Pierre Talbot, David Cachera, Eric Monfroy, and Charlotte Truchet. 2019. Combining constraint languages via abstract interpretation. In *Proceedings - International Conference on Tools with Artificial Intelligence, ICTAI*, Vol. 2019-November. IEEE. https://doi.org/10.1109/ICTAI.2019.00016

[29] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: A new approach to optimization. In *ACM SIGPLAN Notices*, Vol. 44. Association for Computing Machinery.

[30] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and extensible equality saturation. In *Proceedings of the ACM on Principles of Programming Languages*, Vol. 5. https://doi.org/10.1145/3434304

Received 2023-03-10; accepted 2023-04-21

Static Analysis of Data Transformations in Jupyter Notebooks

Luca Negrini
luca.negrini@corvallis.it
Corvallis Srl
Verona, Italy

Guruprerana Shabadi
guruprerana.shabadi@polytechnique.edu
École Polytechnique
Institut Polytechnique de Paris
Paris, France

Caterina Urban
caterina.urban@inria.fr
Inria & ENS | PSL, France
Paris, France

Abstract

Jupyter notebooks used to pre-process and polish raw data for data science and machine learning processes are challenging to analyze. Their data-centric code manipulates dataframes through call to library functions with complex semantics, and the properties to track over it vary widely depending on the verification task. This paper presents a novel abstract domain that simplifies writing analyses for such programs, by extracting a unique CFG from the notebook that contains all transformations applied to the data. Several properties can then be determined by analyzing such CFG, that is simpler than the original Python code. We present a first use case that exploits our analysis to infer the required shape of the dataframes manipulated by the notebook.

CCS Concepts: • **Theory of computation → Program analysis**; **Abstraction**; • **Software and its engineering → Automated static analysis**.

Keywords: Static Analysis, Abstract Interpretation, Data Science, Jupyter Notebooks

ACM Reference Format:
Luca Negrini, Guruprerana Shabadi, and Caterina Urban. 2023. Static Analysis of Data Transformations in Jupyter Notebooks. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '23), June 17, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3589250.3596145

1 Introduction

The ever-increasing usage of data-driven decision processes led to *data science* (DS) and *machine learning* (ML) permeating several areas of everyday life, reaching outside the boundaries of computer science and software engineering. Ensuring correctness of these processes is particularly important when they are employed in critical areas like medicine, public policy, or finance. In contrast to robustness verification of trained ML models [9], data pre-processing of the DS/ML has received little attention. As raw data is often inconsistent or incomplete, pre-processing programs, typically implemented in Jupyter notebooks, apply transformations to polish it to the point where it can be visualized or used for training. Errors and inconsistencies at this stage can silently propagate downwards in the DS/ML chain, leading to incorrect conclusions and below-par models [1].

Verification of notebooks can take different directions, such as detecting data leakages (i.e., sharing of information between the training and test datasets [7]) or warning about the introduction of bias or skews [11]. Regardless, Jupyter notebooks are challenging to analyze: code comes in blocks that can be executed in any order with repetitions, and data is manipulated through calls to a vast number of library functions with complex and possibly overlapping semantics.

This paper proposes an abstract interpretation approach to simplify the implementation of verification techniques for Jupyter notebooks containing DS/ML programs. We propose an abstract domain that tracks transformations made to *dataframes*, that is, the in-memory tables containing the input data, in a unique graph. The latter contains data transformations as nodes that are linked by edges encoding the order in which they are applied. The final graph produced at the end of the analysis is a control flow graph (CFG) containing only dataframe transformations: analyses such as the ones mentioned above can be implemented as fixpoints over this CFG, instead of tackling the more complex Python code.

This paper is structured as follows. Section 2 discusses related work. Section 3 introduces PyLiSA, the analyzer used to evaluate our domain, and LiSA, the framework it relies on. Section 4 defines the abstraction for dataframe values. We then explore a first use-case in Section 5, where we use our abstraction for inferring the shape of the dataframes used by a program. We then conclude with a preliminary experiment on a real DS notebook in Section 6.

2 Related Work

Obtaining formal guarantees on the safety and fairness of ML models has been a subject of recent widespread interest [9].

SOAP '23, June 17, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0170-2/23/06…$15.00
https://doi.org/10.1145/3589250.3596145

Our work builds upon this large ecosystem and proposes a verification framework for the data pre-processing stage of the ML pipeline. The closest body of work is [7] which also analyses DS notebooks along with their peculiar execution semantics and proposes an abstraction to detect data leakage. Our approach towards the shape inference of input data follows the work of [8] and extends it to support inputs to programs which contain datasets. Similarly, our objective of inferring input data usage directly derives from the compound data structure usage analysis presented in [10] and adds the ability to track the usage of selections of datasets. The objective of detecting bias/skew introduction is inspired from the `mlinspect` tool proposed in [4]. This tool builds a directed acyclic graph (DAG) of operations (like filters or projections) applied to the data by analyzing the code and using framework-specific backends (like `scikit-learn`). After analyzing the DAG, it suggests potential sources of bias/skew. Although this is promising, it only places syntactic checks and cannot concretely detect which operations cause these problems. Lastly, [5] is an automated data provenance system for Python. However, it requires executing the code which is not always be feasible when large datasets are involved.

3 LiSA and PyLiSA

LiSA [3, 6] (**Li**brary for **S**tatic **A**nalysis) is a modular framework for developing static analyzers based on the abstract interpretation theory. LiSA analyzes CFGs whose statements do not have predefined semantics: instead, users of the framework define custom statement instances implementing language-specific semantic functions, enabling the analysis of a wide range of programming languages and the development of multilanguage analyses. The analysis infrastructure is partitioned into three main areas: call evaluation, memory modeling and value analysis. Each area corresponds to a separate analysis component, that operates agnostically w.r.t. how the others are implemented. At first, calls are abstracted by the *Interprocedural Analysis*, that leaves the remaining components with call-less programs. Then, memory-related expressions are abstracted by the *Heap Domain*, yielding call- and memory-less programs for the *Value Domain* to analyze. Code parsing and semantics are defined in *Frontends*, that can also provide implementations for LiSA's components. In this paper, we employ the Python frontend PyLiSA[1].

4 The Dataframe Graph Domain

We present an abstraction able to capture the structure of the dataframes manipulated in Python code. Intuitively, we employ a graph structure to keep track of all operations that involve dataframes, with edges encoding the order in which they are performed. The graph thus represents the state of each dataframe at a given program point. Nodes of this graph can be referenced by variables of the program. The latter thus refers to the dataframe corresponding to the sub-graph obtained with a backward DFS starting from the node. We adopt a two-level mapping: program variables point to labels, and the latter are mapped to the nodes. This enables simple handling of dataframe aliasing (i.e., two variables will be mapped to the same label) and updates (i.e., by changing the nodes pointed by a label, we indirectly update all variables pointing to the label).

Our domain $\mathcal{D}^\#$ is meant to be an abstraction of the portion $\mathcal{D}$ of the program state that stores information about dataframes: we rely on an auxiliary domain to reason about the remainder of the state. As, in our experience, non-dataframe values appearing in the notebooks we target are mostly constants, $\mathcal{D}^\#$ cooperates with a simple constant propagation domain $\mathcal{CP}^\#$. We rely on the semantics of $\mathcal{CP}^\#$ to abstract:

- a string expression s as a constant string σ;
- a list of strings cl as a constant list of constant strings $\langle \sigma_1, \ldots, \sigma_n \rangle$;
- a list of dataframes dl as a constant list of abstract labels $\langle \ell_1, \ldots, \ell_n \rangle$ that will be introduced shortly;
- the left-hand side of an assignment x to a set of identifiers $\{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$.

We begin defining $\mathcal{D}^\#$ by introducing the graph structure, where a graph $g^\# = (N, E) \in \mathcal{G}^\#$ is composed by a set of nodes $N \subseteq \mathcal{N}$ and a set of edges $E \subseteq \mathcal{E}$. Elements of $\mathcal{N}$ are:

- `read`(σ), initializing a dataframe with the contents of file σ;
- `access`$(\sigma_1, \ldots, \sigma_k)$, accessing columns $\sigma_1, \ldots, \sigma_k, k \in \mathbb{N}$;
- `transform`(f), transforming values through an auxiliary function f;
- `concat`, concatenating multiple dataframes;
- `filter`(σ, op, v), selecting rows where v^σ op v holds (with v^σ being the value of column σ);
- `assign`$(\sigma_1, \ldots, \sigma_k)$, assigning columns $\sigma_1, \ldots, \sigma_k, k \in \mathbb{N}$ to a new value.

where σ and v are string and value abstractions in $\mathcal{CP}^\#$, $op \in \{=, \neq, >, \geq, <, \leq\}$ and f is the signature of a Python function. Instead, elements of $\mathcal{E}$ are (with $n, n' \in \mathcal{N}, i \in \mathbb{N}$):

- $n \rightarrow n'$ is an edge encoding the sequential order of operations;
- $n \rightsquigarrow_i n'$ is a concatenation edge, where n is the i-th dataframe in the concatenation that builds n' (note that n' can have more incoming concatenation edges using the same i, indicating multiple candidates for the same index);
- $n \twoheadrightarrow n'$ is an assign edge, where n is the right-hand side of the assignment n' (once more, n' can have more incoming assign edges, indicating multiple candidates for the right-hand side).

$\mathcal{D}^\#$ also contains two maps, both relying on abstract labels. A label $\ell \in \mathcal{L}$ is an arbitrary synthetic identifier that serves

[1] https://github.com/lisa-analyzer/pylisa.

```
1 import pandas as pd
2 df1 = pd.read_csv('italy.csv')
3 df1['birth'] = pd.to_datetime(df1['birth'])
4 df2 = pd.read_csv('france.csv')
5 df2 = df2['age' < 50]
6 df3 = pd.concat([df1, df2])
```

Figure 1. Python DS running example

as an abstract name for a set of nodes in $\mathcal{N}$, where $\mathscr{L}$ is the finite set of all possible labels. While we do not impose any specific structure on $\mathscr{L}$, a common characterization of labels is to have one for each program point. $\mathcal{D}^\#$ contains (i) a function $\mathscr{L} \to \wp(\mathcal{N}) \in \mathcal{L}^\#$ from labels to sets of nodes, and (ii) a map $\mathbb{V}ar \to \wp(\mathscr{L}) \in \mathcal{V}^\#$ keeping track of which possible labels a variable can refer to. Notice that, depending on the analyzer's infrastructure, variables can correspond to abstract memory locations or program variables.

We can now define $\mathcal{D}^\#$ as the Cartesian product $\mathcal{V}^\# \times \mathcal{L}^\# \times \mathcal{G}^\#$, that is a complete lattice since $\mathcal{G}^\#$ is a Cartesian product of powersets, and $\mathcal{V}^\#$ and $\mathcal{L}^\#$ are functional lifts of powersets. One concern with infinite lattices such as $\mathcal{D}^\#$ is the convergence of fixpoint iterations over them. As $\mathcal{G}^\#$ intuitively does not satisfy ACC[2], a widening operator is required. As, in our experience, the DS notebooks that this domain targets mostly contain sequential code with very few loops that stabilize in few iterations, we employ a naive widening as $d_1^\# \nabla d_2^\# = d_1^\#$ **if** $d_1^\# = d_2^\#, \top$ **otherwise**. With such an operator we ensure termination of the analysis, and we leave the study of a more precise widening as future work.

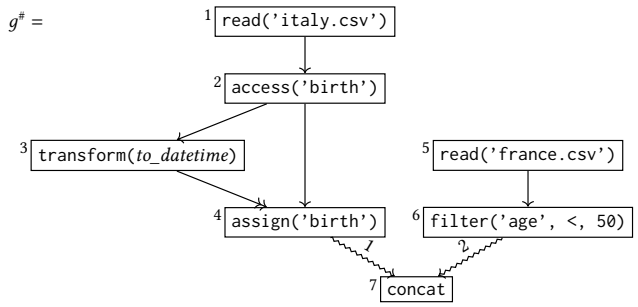
Example. Figure 2 reports the $d^\#$ instance abstracting the code of Figure 1. For the sake of clarity, nodes of $g^\#$ are enriched with a numerical identifier on the top-left corner to easily identify them. Such identifiers are used in the co-domain of $l^\#$ to represent them. We show how this graph is constructed while defining the abstract semantics.

The connection between $\mathcal{D}$ and $\mathcal{D}^\#$ is established by the abstraction function α and the concretization function γ. A set of functions $\{\bar{d}_1, \ldots, \bar{d}_m\}, \bar{d}_i \in \mathcal{D}, 1 \leq i \leq m, m \in \mathbb{N}^\infty$ can be abstracted to an element $d^\# \in \mathcal{D}^\#$ through function $\alpha : \wp(\mathcal{D}) \to \mathcal{D}^\#$, defined as the lub of the abstractions of each individual $\bar{d}$. The abstraction of a single function is defined as $\alpha(\bar{d}) = (v^\#, l^\#, g^\#)$, with:

- $\bar{d} = \{(v_1, d_1), \ldots, (v_k, d_k)\}, 1 \leq i \leq k, k \in \mathbb{N}$;
- $(g_i^\#, n_i) = \mathsf{shape}(d_i)$, $g^\# = \bigcup_i g_i^\#$;
- $l^\# = \{(\ell_1, \{n_1\}), \ldots, (\ell_k, \{n_k\})\}$;
- $v^\# = \{(v_1, \{\ell_1\}), \ldots, (v_k, \{\ell_k\})\}$.

The abstraction of a single dataframe map exploits $\mathsf{shape} : \mathbb{D} \to \mathcal{G}^\# \times \mathcal{N}$, an auxiliary function that extracts the shape of a concrete dataframe as a single-path graph containing only a `read` node followed by an `access` node reporting all the existing columns, and returning the graph itself and its

[2]As $\mathcal{N}$ and $\mathscr{E}$ are infinite sets, one can keep adding new nodes and edges without the graph ever stabilizing.



$$v^\# = \{(\mathtt{df1}, \{\ell_1\}), (\mathtt{df2}, \{\ell_4\}), (\mathtt{df3}, \{\ell_5\})\}$$
$$l^\# = \{(\ell_1, \{4\}), (\ell_2, \{3\}), (\ell_3, \{5\}), (\ell_4, \{6\}), (\ell_5, \{7\})\}$$

Figure 2. Example $d^\#$ abstracting the code of Figure 1

unique leaf. The abstraction of a set of states is thus the union of the abstraction of each individual state, generated by creating the graph $g^\#$ containing the shape (that is, the access to all columns C optionally preceded by the reading of source S) of all existing dataframes, having each variable refer to the corresponding node in $g^\#$.

As α is *join-preserving*, the concretization function γ can be defined in terms of α, according to Proposition 7 of [2], also inducing the Galois connection $\langle \wp(\mathcal{D}), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{D}^\#, \sqsubseteq \rangle$, where $\sqsubseteq$ is the lift of $\subseteq$ to functions and Cartesian products.

Abstract Semantics. The abstract semantics of $\mathcal{D}^\#$ is defined w.r.t the one of $\mathcal{CP}^\#$, that is used to evaluate non-dataframe expressions. $\mathcal{D}^\#$ evaluates dataframe expressions to a set of labels, identifying nodes representing the dataframes that correspond to the expression. In the following, we give intuitive definitions of the semantics of expressions that involve dataframes.

Assignment. Whenever the right-hand side of an assignment `x = df` is a dataframe expression, $v^\#$ must be updated for the corresponding variable. Specifically, as `df` evaluates to a set of labels $\{\ell_1, \ldots, \ell_w\}$, and `x` evaluates to a set of identifiers $\{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$, $v^\#$ can be updated to $v' = v'[\mathtt{x}_i \mapsto \{\ell_1, \ldots, \ell_w\}], \forall \mathtt{x}_i \in \{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$.

Example. When evaluating the assignment at line 2 of Figure 1, the semantics stores the label pointing to the `read` node (whose creation is dictated by the abstract semantics of *read*). $v^\#$ is thus extended with the pair $(\mathtt{df1}, \{\ell_1\})$, where $\{\ell_1\}$ is the label returned by the semantics of *read*.

Variable evaluation. Whenever a variable is referenced throughout the program, our semantics must evaluate it to the corresponding labels if it refers to a dataframe, while the remaining variables are handled by $\mathcal{CP}^\#$. Thus, when `x` resolves to a dataframe, it evaluates to $v^\#(\mathtt{x})$.

Example. When evaluating line 6 of Figure 1, `df1` and `df2` must be first resolved to the dataframes they represent: `df1` evaluates to $\{\ell_1\}$ while `df2` is evaluated to $\{\ell_4\}$, as the two variables are mapped to $\{\ell_1\}$ and $\{\ell_4\}$ in $v^\#$, respectively.

Dataframe initialization. When dataframes are initialized with the contents of an external resource through *read*, the operation cannot be precisely modeled statically as the contents of the resource are unknown at compile time. We thus symbolically record the source of the data by adding a `read` node to the graph. If the argument of *read* is abstracted by $\mathcal{CP}^\#$ as σ, the semantics adds a new `read`(σ) and returns a unique label $\{\bar{\ell}\}$ that points to the freshly added node.

Example. When the `read` call at line 2 of Figure 1 is evaluated, $g^\#$ is still empty. A single `read('italy.csv')` node is added to $g^\#$ by the semantics, that then extends $l^\#$ with a fresh label ℓ_1, that is mapped to a set containing the node itself, and is used as the only label in the result of the evaluation.

Column access. The *access* transformation accesses a set of columns of the target dataframe. As this operation does not create a new dataframe, it is modeled as an in-place operation, directly affecting the nodes pointed by the labels of its argument: the semantics adds an `access`$(\sigma_1, \dots, \sigma_n)$ node (where $\sigma_1, \dots, \sigma_n$ is the $\mathcal{CP}^\#$ abstraction for the column list), connecting it to the nodes of the first argument using normal edges $\rightarrow$ and mapping it to a new label.

Example. When the access to column `birth` at line 3 of Figure 1 is evaluated, `df1` is first processed, producing $\{\ell_1\}$ as abstract value. Then, the evaluation of the column names yields $\langle$`'birth'`$\rangle$, resulting in the creation of node `access('birth')`. The graph is then extended adding (i) the newly created node with id 2, and (ii) a normal edge connecting nodes 1 and 2. Furthermore, ℓ_1 is remapped to $\{2\}$ inside the resulting $l^\#$ (the mapping is not visible in Figure 2 as evaluation of following statements overwrites it).

Value transformation. Tracking transformations of dataframe values can be problematic, as the functions carrying the transformation must be summarized somehow. Instead, we provide a lightweight semantics that records the signature of the used function inside a node of the graph, deferring further reasoning to successive abstractions. As this is not an in-place operation, the semantics creates a new label that is mapped to the transformation node, and the latter is connected to nodes representing the target dataframe (thus branching off the original dataframe). The label is then returned as the unique result of the evaluation. The new `transform`(f) node is also connected to the nodes of the receiving dataframe using normal edges $\rightarrow$.

Example. When the `to_datetime` call at line 3 of Figure 1 is evaluated, our semantics first evaluates the column access, producing $\{\ell_1\}$ as shown earlier. The semantics then (i) creates a unique `transform(to_datetime)` node, (ii) adds it to the graph with id 3, and (iii) connects it to node 2 with a normal edge. The mapping between ℓ_2, a fresh and unused label, and the singleton set containing node 3 is also introduced in $l^\#$, and $\{\ell_2\}$ is returned as the evaluation's result.

Dataframe assignment. When a portion of a dataframe is overwritten with new values, the semantics of variable assignment cannot be employed, as no variable changes value. Instead, the semantics of *assign* records the assignment as an `assign`$(\sigma_1, \dots, \sigma_n)$ node in the graph (with $\sigma_1, \dots, \sigma_n$ is the abstraction of the column list produced by $\mathcal{CP}^\#$), connected to both the dataframe receiving it (using normal edges $\rightarrow$) and the value being stored (using assign edges $\twoheadrightarrow$).

Example. When the assignment at line 3 of Figure 1 is evaluated, `df1` is evaluated to $\{\ell_1\}$, the column names evaluate to the constant list $\langle$`'birth'`$\rangle$, and the *transform* expression is resolved to $\{\ell_2\}$. The semantics proceeds by (i) creating an `assign('birth')` node, (ii) adding it to the graph with id 4, (iii) connecting it to node 3 (image of ℓ_2 in $l^\#$) with an assign edge, and finally (iv) connecting it to node 2 (image of ℓ_1 in $l^\#$) with a normal edge. To conclude, ℓ_1 is remapped to a set containing node 4 in $l^\#$, as the assignment has side-effects.

Rows filtering. Similarly to value transformations, abstracting rows filters can be problematic, as different facts about the filtering conditions can be of interest depending on the target analysis. We thus record the condition as-is within a `filter`(σ, op, v) node, delegating its interpretation to successive abstractions. Note that this is not an in-place operation: original dataframes are not modified, but a filtered view of them is returned by the operation. We reflect this in our formalization by yielding a fresh label pointing to the filtering node. The semantics connects the filter node to the ones of the receiving dataframe with normal edges $\rightarrow$.

Example. When rows are filtered at line 5 of Figure 1 is evaluated, the semantics first evaluates its arguments: `df2` is evaluated to $\{\ell_3\}$, the column name evaluates to `'age'`, and the value used for the comparison to the constant `50`. A `filter('age', <, 50)` node is then created, and it is added the graph with id 6. A normal edge connecting it to node 5 (image of ℓ_3 in $l^\#$) is also introduced, and the label ℓ_4 is created and mapped to a set containing node 6 in $l^\#$.

Concatenation. The semantics of the concatenation must create a new dataframe with the contents of all of its arguments, that come compacted into a list. For of our domain, this means connecting all nodes of each argument to a `concat` node, that will be the image of a new label, using concatenation edges $\rightsquigarrow$ with the corresponding indexes.

Example. When the `concat` call at line 6 of Figure 1 is evaluated, the semantics first evaluates the list of target dataframes, producing $\langle\{\ell_1\}, \{\ell_4\}\rangle$ as abstraction for the elements of the list. The semantics then (i) creates the `concat` node, (ii) adds it to the graph with id 7, and (iii) connects it to nodes 4 and 6 (images of ℓ_1 and ℓ_4 in $l^\#$) with concatenation edges indexed with 1 and 2. A fresh label ℓ_5 is generated, and is mapped to a singleton set containing node 7 in $l^\#$.

5 A First Application: Shape Inference

Tracking dataframe transformations through $\mathcal{D}^\#$ is just the beginning. In general, successive analyses targeting $\mathcal{D}^\#$ instead of starting from the Python code can still be formalized

and implemented as abstract interpretations. Fixpoint algorithms can be applied over instances of $\mathcal{G}^\#$, treating the nodes as code. In this context, one has to define the abstract semantics w.r.t. each node's meaning, possibly taking into account the edges attached to them. In this section, we explore one of the possible objectives in the analysis of DS programs, as we aim at inferring the shape of each dataframe read from an external source.

We start by defining the domain $C^\#$ of columns. Denoting as $\overline{\Sigma} = \Sigma^* \cup \{\top, \bot\}$ the set possible string abstractions provided by $\mathcal{CP}^\#$, elements of $C^\#$ are functions $\wp(\overline{\Sigma}) \to (\wp(\overline{\Sigma}) \times \wp(\overline{\Sigma}))$ whose domain is composed by sets of strings $\{\sigma_1, \ldots, \sigma_p\}$ representing sources of data, and whose co-domain is built over pairs of sets $\{\dot{\sigma}_1, \ldots, \dot{\sigma}_n\}, \{\hat{\sigma}_1, \ldots, \hat{\sigma}_m\}$. Each $\dot{\sigma}_i$ is a column that is accessed before being assigned (and thus must be part of the original dataframe to prevent runtime errors), and each $\hat{\sigma}_i$ is a column that is assigned during the execution (and thus might not exist in the original dataframe). The columns domain thus aims at inferring, for each external source of data, what columns must exist for the program to not crash. In our abstraction, we consider sets of sources as function keys since dataframes can be created through concatenation, and can thus have multiple sources. In this case, all accessed columns must exist in at least one source. Note that $C^\#$ is a complete lattice, as it is a functional lift of a Cartesian product of powersets.

We informally define the semantics of $C^\#$ w.r.t. nodes of $\mathcal{G}^\#$, as this kind of analysis does not need variable information stored in $\mathcal{L}^\#$ and $\mathcal{V}^\#$. The semantics relies on the auxiliary function `sources` : $\mathcal{N} \to \wp(\overline{\Sigma})$ that extracts all sources (i.e., arguments of `read` nodes) whose data is used to build the dataframe identified by the given node.

Read. When reading data from an external resource, no particular column is accessed. Instead, we define an entry in our state corresponding to the possible abstractions of the resource identifier.

Concat. Similarly to `read`, `concat` does not access any column, but instead introduces a new entry in the post-state corresponding to the union of its sources.

Transform. A transformation does not access any column explicitly, as it operates on the entirety of the dataframe that receives the transformation. As such, its semantics is defined as the identity function.

Access. The column access is the main vector for referencing columns by-name. This is reflected by its semantics, that adds every column name to the left-most set of the corresponding sources if we have no evidence of it being defined earlier.

Filter. As rows filtering is expressed as a condition over the value of a specific column, it indirectly represents a column access. This is once more reflected as the addition of the column name to the left-most set of the corresponding sources, if it was not defined before.

Assign. The dataframe assignment is the only node kind that can safely define non-existing columns. The semantics of this node adds the abstract column names to the right-most set of the corresponding sources.

Example. We now use $C^\#$ to infer the shape of the dataframes abstracted by the graph in Figure 2. For the sake of clarity, we first analyze the left-most branch of the graph as a whole, followed by the right-most one, concluding the analysis with the `concat` node. The analysis begins applying the semantics of `read` to node 1, using an empty $C^\#$ instance $c_0^\# = \{\}$ as pre-state, and producing the entry for the read resource $c_1^\# = \{(\{\texttt{'italy.csv'}\}, (\emptyset, \emptyset))\}$. This is in turn used as pre-state for the evaluation of node 2, where the semantics of `access` populates the function with the accessed column, producing $c_2^\# = \{(\{\texttt{'italy.csv'}\}, (\{\texttt{'birth'}\}, \emptyset))\}$. As the semantics of `transform` is the identity function, the pre-state of node 4 is the join of the post-state of both predecessors: $c_2^\# \dot{\cup} c_2^\# = c_2^\#$. When such result is used as argument for the `assign` semantics, it produces the post-state $c_3^\# = \{(\{\texttt{'italy.csv'}\}, (\{\texttt{'birth'}\}, \{\texttt{'birth'}\}))\}$. Equivalently, the analysis starts with the same empty pre-state $c_0^\#$ at node 5, applying the `read` semantics and producing the post-state $c_4^\# = \{(\{\texttt{'france.csv'}\}, (\emptyset, \emptyset))\}$. Then, $c_4^\#$ is used to compute the result of the `filter` semantics at node 6, that is $c_5^\# = \{(\{\texttt{'france.csv'}\}, (\{\texttt{'age'}\}, \emptyset))\}$. Lastly, at node 7, the pre-state is built as the lub of the predecessors' post-states $c_6^\# = c_3^\# \dot{\cup} c_5^\#$ that thus contains both entries. The semantics of `concat` can then be applied to $c_6^\#$, producing the final state of the analysis:

$$c_7^\# = \left\{ \begin{array}{l} (\{\texttt{'italy.csv'}\}, (\{\texttt{'birth'}\}, \{\texttt{'birth'}\})), \\ (\{\texttt{'france.csv'}\}, (\{\texttt{'age'}\}, \emptyset)), \\ (\{\texttt{'italy.csv'}, \texttt{'france.csv'}\}, (\emptyset, \emptyset)) \end{array} \right\}$$

6 An Early Experiment Using PyLiSA

Both $\mathcal{D}^\#$ and $C^\#$ have been implemented in PyLiSA, that analyzes Jupyter notebooks by extracting the Python code from the cells that contains it. Cells are analyzed according to a specific user-defined execution sequence, defaulting to the order in which cells are defined in the notebook.

We provide the semantics of `pandas` library's functions through LiSA's native CFGs. These are special CFGs that contain a single statement. Calls to native CFGs are evaluated by computing the semantics of their unique node, without running additional fixpoints: the node's semantics thus becomes an abstract summary of the modeled function. Specifically, the semantics of native CFGs modeling `pandas` functions converts them to the nodes presented in Section 4. $\mathcal{D}^\#$ has been implemented as a *Value Domain*, directly embedding the constant propagation domain. At the end of the analysis, a $\mathcal{G}^\#$ instance is extracted from the post-state of the last instruction, and a fixpoint is executed over the it using $C^\#$. Warnings are then issued to inform the user about columns accessed and assigned for each data source.

As a first experiment, we selected the *"Coronavirus (COVID-19) Visualization & Prediction"*[3] dataframe, one of the most popular notebooks aggregating data from different sources on Kaggle, a public repository of Jupyter notebooks for DS. The graph produced when analyzing such code is published on a GitHub Gist[4] as it is too large for this manuscript. Note that the implemented analysis supports additional pandas constructs w.r.t. the ones presented in this paper, that have been omitted as they do not contribute further to the intuition behind the domain. In the graph, these take the form of additional node kinds, whose intuitive meaning is explained in the Gist's introduction. The analysis generates the following warnings (where URL of csv files have been trimmed for compactness), correctly identifying all column names that appear in the notebook:
[File: *daily_reports/08-23-2022.csv*] Columns accessed before being assigned: 'Confirmed', 'Province_State', 'Country_Region', 'Incident_Rate', 'Deaths'
[File: *daily_reports_us/08-23-2022.csv*] Columns accessed before being assigned: 'Province_State', 'Testing_Rate', 'Total_Test_Results'
[File: *time_series_covid19_confirmed_global.csv*] Columns accessed before being assigned: 'Country/Region'
[File: *time_series_covid19_deaths_global.csv*] Columns accessed before being assigned: 'Country/Region'

7 Conclusion

This paper presents an abstract interpretation approach to analyze Python programs employed in data science and machine learning. Such programs manipulate dataframes, that is, complex in-memory tables collecting data that can be used to guide decision processes or train machine learning models. We designed an abstract domain that extracts the operations performed over dataframes, building a graph that encodes the order in which they are performed. Such a graph can be the subject of further analyses, inferring several properties such as the *shape* of the dataframes read by the program, or the absence of data leakages between training and testing phases of a machine learning process. As a guiding example of how to exploit our domain, we defined a simple abstract interpretation that computes, for each file read by the source program (and thus present inside the graph), the set of columns that are either accessed before being assigned, or defined through an assignment. We provided an early implementation of both domains in PyLiSA, a LiSA frontend for Python programs.

There are plenty of future directions that our work can take. As this work is still ongoing, the obvious first line of axis is to prove the soundness of the proposed semantics abstractions, followed by an investigation on their completeness to further improve the domain. Besides, inferring the shape of dataframes is not the only useful analysis one can employ for DS software. In fact, after strengthening shape inference to incorporate rows and cell properties, we aim at providing abstractions to detect data leakages and biases. Lastly, we aim at extending the abstraction to more pandas functions, and to further libraries other than pandas itself.

[3] https://www.kaggle.com/code/therealcyberlord/coronavirus-covid-19-visualization-prediction, version 722.
[4] https://gist.github.com/lucaneg/9621f3296b7b47b12c5ee1c52066b3d1.

References

[1] Irene Y. Chen, Fredrik D. Johansson, and David Sontag. 2018. Why is My Classifier Discriminatory?. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 3543–3554. https://proceedings.neurips.cc/paper/2018/file/1f1baa5b8edac74eb4eaa329f14a0361-Paper.pdf

[2] Patrick Cousot and Radhia Cousot. 1992. Abstract interpretation and application to logic programs. *The Journal of Logic Programming* 13, 2 (1992), 103–179. https://doi.org/10.1016/0743-1066(92)90030-7

[3] Pietro Ferrara, Luca Negrini, Vincenzo Arceri, and Agostino Cortesi. 2021. Static Analysis for Dummies: Experiencing LiSA. In *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis* (Virtual, Canada) *(SOAP 2021)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3460946.3464316

[4] Stefan Grafberger, Shubha Guha, Julia Stoyanovich, and Sebastian Schelter. 2021. MLINSPECT: A Data Distribution Debugger for Machine Learning Pipelines. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2736–2739. https://doi.org/10.1145/3448016.3452759

[5] Mohammad Hossein Namaki, Avrilia Floratou, Fotis Psallidas, Subru Krishnan, Ashvin Agrawal, Yinghui Wu, Yiwen Zhu, and Markus Weimer. 2020. Vamsa: Automated Provenance Tracking in Data Science Scripts. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 1542–1551. https://doi.org/10.1145/3394486.3403205

[6] Luca Negrini. 2023. *A generic framework for multilanguage analysis*. Ph. D. Dissertation. Universitá Ca' Foscari Venezia.

[7] Pavle Subotić, Lazar Milikić, and Milan Stojić. 2022. A Static Analysis Framework for Data Science Notebooks. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 13–22. https://doi.org/10.1145/3510457.3513032

[8] Caterina Urban. 2020. What Programs Want: Automatic Inference of Input Data Specifications. *CoRR* abs/2007.10688 (2020). arXiv:2007.10688 https://arxiv.org/abs/2007.10688

[9] Caterina Urban and Antoine Miné. 2021. A Review of Formal Methods applied to Machine Learning. *ArXiv* abs/2104.02466 (2021). https://doi.org/10.48550/arXiv.2104.02466

[10] Caterina Urban and Peter Müller. 2018. An Abstract Interpretation Framework for Input Data Usage. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 683–710. https://doi.org/10.1007/978-3-319-89884-1_24

[11] Ke Yang, Biao Huang, Julia Stoyanovich, and Sebastian Schelter. 2020. Fairness-Aware Instrumentation of Preprocessing Pipelines for Machine Learning, In Workshop on Human-In-the-Loop Data Analytics (HILDA'20). *Workshop on Human-In-the-Loop Data Analytics (HILDA'20)*. https://par.nsf.gov/biblio/10182459

Received 2023-03-10; accepted 2023-04-21

Speeding up Static Analysis with the Split Operator

Vincenzo Arceri
vincenzo.arceri@unipr.it
University of Parma
Parma, Italy

Greta Dolcetti
greta.dolcetti@studenti.unipr.it
University of Parma
Parma, Italy

Enea Zaffanella
enea.zaffanella@unipr.it
University of Parma
Parma, Italy

Abstract

In the context of static analysis based on Abstract Interpretation, we propose a new abstract operator modeling the *split* of control flow paths: the goal of the operator is to enable a more efficient analysis when using abstract domains that are computationally expensive, having no effect on precision. Focusing on the case of conditional branches guarded by numeric linear constraints, we provide a preliminary experimental evaluation showing that, by using the split operator, we can achieve significant efficiency improvements for a static analysis based on the domain of convex polyhedra. We also briefly discuss the applicability of this new operator to different, possibly non-numeric abstract domains.

CCS Concepts: • **Theory of computation → Program analysis**; • **Software and its engineering → Automated static analysis**.

Keywords: Abstract Interpretation, Static Analysis, Abstract Operators

ACM Reference Format:
Vincenzo Arceri, Greta Dolcetti, and Enea Zaffanella. 2023. Speeding up Static Analysis with the Split Operator. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '23), June 17, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3589250.3596141

1 Introduction

In static analysis tools based on Abstract Interpretation (AI) it is common to define a neat separation between the AI framework and the abstract domains: the two components communicate by invoking abstract domain operators, such as meet, join, widening and narrowing, whose results are combined to correctly characterize the abstract semantics of the program. Among the operations needed during static analysis, one allows to *filter* an abstract element $A \in \mathbb{A}$ over a given predicate $p \in \text{Pred}$, returning a refined abstract element A_p approximating those states of A that satisfy p:

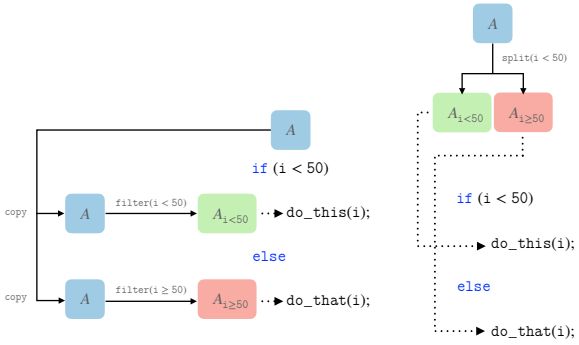
$$\text{filter} \colon \mathbb{A} \times \text{Pred} \to \mathbb{A}.$$

This operator goes by different names depending on the context. Abstract domains adopt a rather specific terminology, which often highlights the kind of predicates that are supported; for instance, the numeric domains in Apron [18] provide `meet_lincons_array` for filtering on a set of linear constraints; similarly, we can find `add_cons` for the polyhedra domains in the PPLite library [8] and `add_congruences` for the grid domain [2] in the PPL library [4]. Static analysis tools usually adopt a more generic name, sometimes promoting the operator to an abstract instruction in their program representation: for instance, we have `Comparison` in IKOS AR [9], `S_assume` in MOPSA [20] and `assume` in both Clam/Crab [16] and LiSA [12]. The filter operator can be used to enforce properties that are known to hold at a specific program point: for instance, after a call to a mock library function, we can filter the abstract state on the post-condition of the function. The same operator is also commonly used to model the conditional split of the control flow of the program. For instance, the code fragment

```
if (i < 50) do_this(i); else do_that(i);
```

can be modeled as shown in Figure 1a: the input abstract element A is cloned to obtain two copies; these are then filtered on the predicate $p = (i < 50)$ for the `then` branch and on its complement $\neg p = (i \geq 50)$ for the `else` branch.

This implementation approach is both correct and precise, but in some contexts could incur avoidable inefficiencies, as it



(a) Using filter operator **(b)** Using split operator

Figure 1. Abstract modeling of a conditional branch.

SOAP '23, June 17, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0170-2/23/06…$15.00
https://doi.org/10.1145/3589250.3596141

replicates most of the work done to evaluate the complementary predicates in the two program branches. The overhead is probably negligible, hence acceptable, as long as considering the most efficient abstract domains, such as the domain of intervals [10]. Things might be different when adopting more expensive and precise abstract domains, such as the domain of convex polyhedra [11]; in these cases, it might be worth exploring alternative implementation strategies to better preserve efficiency. In particular, in Figure 1b we show an alternative approach where the conditional branch is modeled using the *split* operator:

$$\text{split} : \mathbb{A} \times \text{Pred} \to \mathbb{A} \times \mathbb{A}.$$

This new abstract domain operator, initially proposed in [7], filters the input abstract element on a predicate p and on its complement $\neg p$ *at the same time*, allowing for the factorization of any replicated computational effort. Note that, if no optimization is possible, the abstract domain can just resort to the default implementation, which clones the abstract element and invokes (twice) the filter operator.

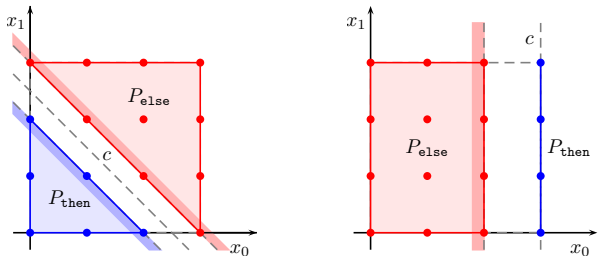
While this idea is both simple and intuitively promising, to the best of our knowledge its effectiveness has never been evaluated in the context of classical AI-based static program analysis:[1] the goal of the current work is to provide such an evaluation. To this end, we first describe the design changes that are required, both at the interface and at the implementation levels, in order to accomodate the new split operator into an existing static analysis tool. Then, we show the results of our preliminary experiments, which focus on splits defined on numerical predicates and on the abstract domain of convex polyhedra.

In principle, the high level specification of the split operator allows for a general adoption, independently from the considered abstract domain. However, since its end goal is to enable efficiency improvements (leaving correctness and precision unaffected), its actual effectiveness necessarily depends on *profitability* considerations (e.g., the computational cost of filtering on a specific class of predicates and the frequency of these operations in a typical analysis). Hence, we will also briefly discuss the applicability of the approach to more general contexts.

2 Splitting Polyhedra on Linear Constraints

In this section we briefly discuss how a split operator defined on a numerical predicate can be efficiently implemented on the domain of convex polyhedra [11]. For space reasons, we will only *hint* at the optimizations, without a proper explanation of their details: these are available in release 0.10.1 of the open source library PPLite [5, 6, 8].

The split operator proposed in [7] for the domain of convex polyhedra focuses on the case of *rational splits*: since in this case the constraints added to the `then` and `else` branches are complementary, the new abstract operator can factorize most of the work, leading to a rather systematic halving of the computational effort.[2] However, often the predicate guarding a conditional branch is defined on variables having an integral datatype: in this case the use of a rational split produces a safe result, but typically incurs a precision loss. Therefore, we now focus on *integral splits*. Consider first the case of a split based on a (strict or non-strict) linear inequality constraint.



(a) Split on $c \equiv (2x_0 + 2x_1 \leq 5)$ **(b)** Split on $c \equiv (x_0 = 3)$

Figure 2. Examples of integral splits.

Example 2.1. Let $P \equiv \{0 \leq x_0 \leq 3, 0 \leq x_1 \leq 3\}$ be a polyhedron in $\mathbb{R}^2$ and consider a conditional branch guarded by constraint $c \equiv (2x_0 + 2x_1 \leq 5)$. If the variables are known to be integral, this guard can be refined to $c_{\texttt{then}} \equiv (x_0 + x_1 \leq 2)$; the corresponding integral complement constraint is $c_{\texttt{else}} \equiv (x_0 + x_1 \geq 3)$. Hence, we obtain $P_{\texttt{then}} \equiv \{0 \leq x_0, 0 \leq x_1, x_0 + x_1 \leq 2\}$ and $P_{\texttt{else}} \equiv \{x_0 \leq 3, x_1 \leq 3, x_0 + x_1 \geq 3\}$, shown in Figure 2a.

The two constraints $c_{\texttt{then}}$ and $c_{\texttt{else}}$ are not complementary when considered in the real relaxation $\mathbb{R}^2$ and hence the optimized implementation of the integral split operator cannot be as effective as that for the rational case. However, since the two constraints have the same slope, the implementation can still factor out most of the (uselessly duplicated) work done when computing the scalar products of each generator of P with the two constraints, reducing the overall computational cost. Note that the described integral refinement process is generally incomplete: there are cases where one of the branches has no integral solution, but the abstract domain is unable to detect it due to the real relaxation (unless performing further expensive checks).

We now consider the case of an integral split based on a linear (dis-) equality constraint. This case turns out to be trickier, since the domain of polyhedra, like most numerical domains based on convex approximations, is unable to precisely filter on a disequality constraint.

Example 2.2. For polyhedron P of Example 2.1, consider a branch guarded by constraint $c \equiv (x_0 = 2)$. We can be

[1] The experimental evaluation reported in [7] targets the analysis of a particular class of hybrid systems.

[2] Readers interested in the details of the optimized implementation of rational splits are referred to [7, Section 5].

precise on the equality branch $P_{\texttt{then}} \equiv \{x_0 = 2, 0 \leq x_1 \leq 3\}$; on the other branch we may try to lower the disequality into a pair of inequalities, intuitively computing $P^{<}_{\texttt{else}} \equiv \{0 \leq x_0 \leq 1, 0 \leq x_1 \leq 3\}$ and $P^{>}_{\texttt{else}} \equiv \{x_0 = 3, 0 \leq x_1 \leq 3\}$; unfortunately, this effort towards precision is later made useless by the join computation $P_{\texttt{else}} = P^{<}_{\texttt{else}} \uplus P^{>}_{\texttt{else}} = P$.

Some attempts can be made to identify those lucky cases where a disequality can be successfully refined into an inequality constraint, as in the following example.

Example 2.3. When splitting polyhedron P of Example 2.1 on constraint $c \equiv (x_0 = 3)$, we obtain $P_{\texttt{then}} \equiv \{x_0 = 3, 0 \leq x_1 \leq 3\}$, $P^{<}_{\texttt{else}} \equiv \{0 \leq x_0 \leq 2, 0 \leq x_1 \leq 3\}$ and $P^{>}_{\texttt{else}} \equiv \bot$; hence, as shown in Figure 2b, $P_{\texttt{else}} = P^{<}_{\texttt{else}} \uplus P^{>}_{\texttt{else}} = P^{<}_{\texttt{else}}$.

The approach varies depending on the considered static analysis tool. For instance, Crab is able to detect a lucky case when the disequality constraint $c^{\neq}$ satisfies rather specific conditions:

(a) $c^{\neq} \equiv (x_i \neq x_j)$ and P implies either $c^{\leq} \equiv (x_i \leq x_j)$ or $c^{\geq} \equiv (x_i \geq x_j)$; or
(b) $c^{\neq} \equiv (x_i \neq \mathit{expr})$ and there exists $k \in \mathbb{Z}$ such that P implies $c_k \equiv (\mathit{expr} = k)$,[3] and P also implies either $c^{\leq} \equiv (x_i \leq k)$ or $c^{\geq} \equiv (x_i \geq k)$.

What should be noted here is that, in the lack of a specific operator for the integral split, the analysis tool is forced to perform several calls to lower level abstract operators (entailment checks, evaluations of the value range of a linear expression, etc.), with a corresponding multiplication of the overheads that are inherently incurred when interfacing the static analysis tool with a *generic* abstract domain component; hence, the more precise (and expensive) domains will likely witness a degradation of their overall efficiency.

3 Enabling Splits in a Static Analysis

For our experiments we have chosen Clam/Crab, which is the Abstract Interpretation engine included in the SeaHorn framework [15]. The Clam component uses Clang/LLVM to obtain the LLVM bitcode of the program under analysis and then generates the corresponding CrabIR representation [16]; this is processed by the Crab component, which computes the abstract semantics according to the chosen analysis configuration. The latter includes, among many other parameters, the choice of the abstract domain: Crab supports many (combinations of) abstract domains and includes interfaces towards the domains provided by libraries Apron [18] and ELINA [22].

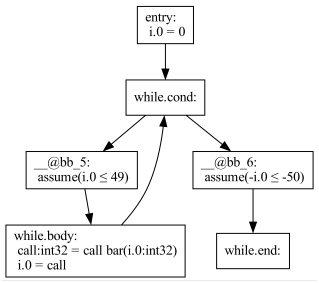
The addition of a new abstract operator requires that suitable changes are applied to both the abstract domain and the fixpoint engine components of the static analysis tool.

[3] I.e., all the variables x_j occurring in *expr* are bound in P to constant values.

Changes in the abstract domain component. We have first implemented the rational and integral split operators for the polyhedra domains included in the PPLite library [5, 6, 8] and we made them available (by adding a non-generic, *ad hoc* function) in the corresponding Apron interface wrapper. Then, we have extended the generic abstract domain interface in Crab by adding a new method for the split operator: this invokes the new abstract operator when the interface is instantiated with a PPLite domain, while resorting to the unoptimized implementation (based on the filter operator) when it is instantiated with the other domains.

```
void foo() {int i=0; while(i<50) i=bar(i);}
```

(a) A simple C function.



(b) Original CrabIR CFG. **(c)** CrabIR CFG with split.

Figure 3. The addition of split statements in CrabIR.

Changes in the fixpoint approximation engine. The adaptation of the fixpoint engine required more work than expected. This is mainly due to a CrabIR language design choice (inherited from IKOS AR form [9]) whereby all conditional branches are expressed in a declarative way, combining a non-deterministic branch with the addition, in each target of the branch, of `assume` abstract statements encoding the branch condition (see the CFG in Figure 3b for an example); roughly speaking, the unoptimized implementation of the conditional branch is *hard coded* in the CrabIR representation, preventing the application of the split operator. As a workaround, we enriched CrabIR by adding a new abstract statement, called `split`, and then modified Clam to generate this new statement whenever translating a numerical conditional branch (see the CFG in Figure 3c). Branches based on Boolean and pointer tests are not affected and hence maintain the declarative encoding.

The other main change to the fixpoint computation engine regards the choice of where to store the invariants computed during the analysis. Exploiting the declarative encoding of conditional branches, by default Crab annotates each node in the CFG with the pairs $\langle \mathit{pre}, \mathit{post} \rangle$ of invariants that are valid at the start (*pre*) and at the end (*post*) of the node. However, when the CFG is modified by introducing `split` statements this approach is no longer adequate, because we would need

to store two different invariants ($post_{\texttt{then}}$ and $post_{\texttt{else}}$) at the exit of those nodes ending with a `split` statement. Therefore, we modified the engine so as to store an invariant along each *edge* of the CFG, as well as storing an invariant on those nodes where a widening/narrowing is computed. At the end of the analysis, in a *finalization phase*, the edge invariants are used to compute the $\langle pre, post \rangle$ pairs of each CFG node, which requires computing many joins; while this simplistic approach incurs avoidable inefficiencies, it preserves the expectations of the other Clam/Crab components, that have not been modified.

It is worth noting that our current prototype is considering a *subset* of all the numerical branches that, in principle, could benefit from the split operator. Namely:

- *rational* splits are disregarded because they only make sense when branching on a floating point condition; currently, Clam/Crab provides limited support for floating point datatypes and safely ignores these conditions, yielding a pure non-deterministic branch;
- similarly, Clam/Crab safely ignores integral splits when the corresponding predicates are linear inequalities defined on *unsigned* variables; this is done to avoid potential safety issues related to the wrap-around semantics as defined in C-like languages;
- finally, our prototype currently disregards those *implicit* numerical branches encoded in CrabIR `select` statements, which implement conditional assignments.

As a consequence we conjecture that, in our experimental evaluation, the efficiency improvements obtainable thanks to the split operator are underestimated.

4 Experimental Evaluation

Our prototype analyzer was obtained by modifying the `dev14` branch of Clam/Crab,[4] which is based on LLVM 14. In our experiments we tried to apply minimal changes to the default configuration of the analyzer: in particular, we instructed LLVM to systematically inline function calls, so as to improve the call context sensitivity of the analysis. Note that, by default, Clam/Crab instructs LLVM to lower all switch statements, which are thus translated to chains of conditional branches and hence can benefit of the split optimization.

Table 1 reports the timing results for some of the tests considered in our *preliminary* experimental evaluation. We analyzed programs coming from two different sources. First, we considered 39 C source files distributed with PAGAI [17] which are variants of benchmarks taken from the SNU real-time benchmark suite for WCET (worst-case execution time) analysis; the results for the 5 tests whose analysis time was greater than a second are shown in the top half of the table. Then, we enriched our benchmark suite by also considering a few Linux drivers from the SVCOMP repository;[5] in this case, we applied no specific selection criterion and just cherry-picked a few drivers, shown in the lower half of the table, having reasonable analysis time.

Table 1. Overall static analysis time without/with splits.

(abridged) test name	without splits PPLite	without splits ELINA	with splits PPLite	time ratio
adpcm	108.2	(⋆) 1.6	105.9	1.02
prog9000	31.8	241.1	42.0	0.76
nsichneu	30.9	40.8	21.1	1.46
decompress	5.3	5.5	2.3	2.30
filter	2.5	2.3	2.5	1.00
mmc-host	487.1	435.1	412.9	1.18
firewire	98.7	(⋆) 92.0	22.7	4.35
hwmon-abituguru3	87.5	91.0	52.8	1.66
media-usb-tm6000	23.2	22.3	12.4	1.87
media-pci-ttpci	12.3	12.9	10.8	1.14
power-bq2415x	10.1	10.9	11.6	0.87
hid-usb	8.6	10.1	6.0	1.43

The 2nd column in Table 1 shows the baseline for the efficiency evaluation, i.e., the overall analysis time in seconds when using `F_Poly`, the Cartesian factored convex polyhedra domain of PPLite; note that we include time spent in pre-analysis phases (e.g., parsing, LLVM bitcode generation and Clam preprocessing steps), while excluding post-analysis phases (e.g., assertion checks based on the results of the analysis).[6] In the 3rd column we show the time obtained when using the Cartesian factored convex polyhedra domain of ELINA [22]: this is done to highlight that our starting point for the efficiency comparison is in line with what is considered the most efficient library for convex polyhedra. Note however that ELINA cannot be used as a proper baseline, as it is well known that, by using machine integers (rather than the arbitrary precision integers adopted by PPLite), it sometimes raises overflow exceptions, after which it returns an over-approximation of the actual result, hence affecting both the efficiency and the precision of the analysis; these cases are highlighted in the table using (⋆).

The 4th column of the table reports the overall analysis time obtained when enabling the split optimization in our prototype, while the last column shows the speedup with respect to the baseline. On the considered tests, we obtain significant speedups, sometimes beyond our own expectations. In a few cases we also obtain slowdowns: these seem to be mainly caused by the unoptimized invariant finalization phase described in the previous section; by disabling this phase (leaving invariants on CFG edges), the analysis time of tests prog9000 and power-bq2415x can be reduced by 9.1 and 3.2 seconds, respectively.

A remarkable side effect of the split optimization is a significant reduction in peak memory usage, as highlighted by the data shown in Table 2. The 2nd and 5th columns of the

[4] https://github.com/seahorn/clam/tree/dev14
[5] https://github.com/sosy-lab/sv-benchmarks/c/ldv-linux-4.2-rc1

[6] Experiments have been performed on a laptop with an Intel Core i7-3632QM CPU, 16 GB of RAM and running GNU/Linux 5.15.0-60.

Table 2. Number of nodes, splits and maximum RSS.

	without splits			with splits		mem
(abridged) test name	nodes	mem	splits	nodes	mem	ratio
adpcm	146	87	34	93	85	1.01
prog9000	1491	1629	275	947	1415	1.15
nsichneu	2004	1591	625	754	628	2.53
decompress	1032	177	266	638	87	2.15
filter	1121	263	187	809	254	1.04
mmc-host	26254	9705	3772	19701	1102	8.81
firewire	9925	7690	2038	6842	282	27.27
hwmon-abituguru3	2653	271	521	2088	130	2.08
media-usb-tm6000	15447	3300	1453	12886	200	16.50
media-pci-ttpci	9057	524	156	5610	147	3.56
power-bq2415x	23763	351	1985	20518	256	1.37
hid-usb	7303	185	1478	4704	119	1.55

table show the number of nodes in the CrabIR CFGs generated without and with the split optimization (note that, in both cases, we refer to the CFGs *after* the LLVM inlining phase). As intuitively suggested by the CFGs in Figure 3, the decrease in the number of nodes is directly caused by the introduction of the numerical `split` statements, whose number is reported in the 4th column. In the 3rd and 6th columns of Table 2 we provide an estimation of the peak memory usage by reporting the maximum RSS (Resident Set Size) for the two aforementioned configurations (in MB); the last column of the table shows the ratio of the two memory measurements. A recent paper [19] proposes a technique to reduce the memory footprint of the static analysis tool IKOS (which shares with Crab the main design of the fixpoint approximation engine). Since the technique adopted in [19] is completely independent from the split optimization, we conjecture that the two optimizations can be applied together, combining their improvements.

Our experimental evaluation also confirmed that the abstract execution of the split statements using the new split abstract domain operator has no effect on precision: using the `clam-diff` helper tool, we observed no regression when systematically comparing the invariants produced by the baseline and the optimized analyses.

5 Conclusions

This paper proposes a new abstract domain operator that is able to speed up the static analysis when splitting the control flow path on a predicate and its complement. Even though our current prototype can only handle a subset of the control flow splits of the program, it is able to obtain non-trivial memory and time improvements on several tests, including both synthetic benchmarks and more realistic programs.

Future work can investigate several directions. First of all, the current prototype can be extended to enable the optimization on more kinds of numerical split statements, e.g., the correct handling of branches on unsigned integral expressions and the implicit branches in `select` statements.

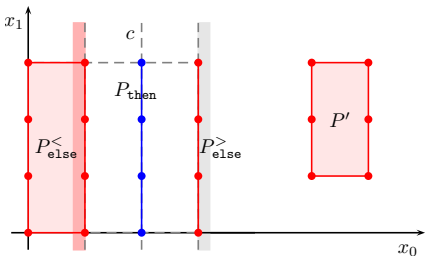


Figure 4. Example of split on a powerset domain.

Second, we plan to evaluate the applicability of the approach to other abstract domains; as briefly discussed before, while the overall idea is quite general, its effectiveness depends on profitability considerations. In particular, we believe that abstract domains supporting the representation of *disjunctive* information are promising candidates for an optimized split operator. As a first example, one could consider the finite powerset of convex polyhedra [3]. Note that, when using a disjunctive domain, it is clearly possible to workaround the limitations of convex over-approximations and thus improve the precision of splits. For instance, in Figure 4 we consider the split of the powerset $S = \{P, P'\}$ on the equality constraint c, where P and c are those described in Example 2.2; hence, by avoiding the convex polyhedral hull approximation, we can obtain $S_{\texttt{then}} = \{P_{\texttt{then}}\}$ and $S_{\texttt{else}} = \{P^{<}_{\texttt{else}}, P^{>}_{\texttt{else}}, P'\}$; note that P', which is not directly affected by the split operator, can be simply "moved" into $S_{\texttt{else}}$, avoiding useless and costly copy operations.

Other disjunctive abstract domains that are probably worth considering are LDDs [14] and RDDs [13] (Linear and Range Decision Diagrams), both available in Crab. We also plan to investigate the application of split operators to non-numeric domains, such as ROBDD-based domains for Boolean formulae or DFA-based abstract domains for string analysis [1, 21]; for instance, we could explore an optimized split operator for conditional branches based on string predicates, such as `str.startsWith("prefix")`, whose default implementations on the domain of DFAs are often expensive.

Finally, it would be interesting to extend our experimental evaluation to different static analysis tools; while we conjecture that similar results can be obtained for other tools analysing the low level program representation (e.g., IKOS [9] and PAGAI [17]), it is more difficult to predict the effectiveness of the optimization for those tools targeting the AST-based high level representations (e.g., MOPSA [20]).

Acknowledgments

The authors would like to express their gratitude to Jorge A. Navas for his help in getting them acquainted with the inner workings of Clam/Crab and for developing helper tool `clam-diff` to compare the precision of different analyses.

References

[1] Vincenzo Arceri, Isabella Mastroeni, and Sunyi Xu. 2020. Static Analysis for ECMAScript String Manipulation Programs. *Appl. Sci.* 10 (2020), 3525. https://doi.org/10.3390/app10103525

[2] Roberto Bagnara, Katy Louise Dobson, Patricia M. Hill, Matthew Mundell, and Enea Zaffanella. 2006. Grids: A Domain for Analyzing the Distribution of Numerical Values. In *Logic-Based Program Synthesis and Transformation, 16th International Symposium, LOPSTR 2006, Venice, Italy, July 12-14, 2006, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 4407)*, Germán Puebla (Ed.). Springer, 219–235. https://doi.org/10.1007/978-3-540-71410-1_16

[3] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. 2007. Widening operators for powerset domains. *Int. J. Softw. Tools Technol. Transf.* 9, 3-4 (2007), 413–414. https://doi.org/10.1007/s10009-007-0029-y

[4] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. 2008. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.* 72, 1-2 (2008), 3–21. https://doi.org/10.1016/j.scico.2007.08.001

[5] Anna Becchi and Enea Zaffanella. 2018. A Direct Encoding for NNC Polyhedra. In *Computer Aided Verification - 30th International Conference, CAV 2018, Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10981)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 230–248. https://doi.org/10.1007/978-3-319-96145-3_13

[6] Anna Becchi and Enea Zaffanella. 2018. An Efficient Abstract Domain for Not Necessarily Closed Polyhedra. In *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11002)*, Andreas Podelski (Ed.). Springer, 146–165. https://doi.org/10.1007/978-3-319-99725-4_11

[7] Anna Becchi and Enea Zaffanella. 2019. Revisiting Polyhedral Analysis for Hybrid Systems. In *Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11822)*, Bor-Yuh Evan Chang (Ed.). Springer, 183–202. https://doi.org/10.1007/978-3-030-32304-2_10

[8] Anna Becchi and Enea Zaffanella. 2020. PPLite: Zero-overhead encoding of NNC polyhedra. *Inf. Comput.* 275 (2020), 104620. https://doi.org/10.1016/j.ic.2020.104620

[9] Guillaume Brat, Jorge A. Navas, Nija Shi, and Arnaud Venet. 2014. IKOS: A Framework for Static Analysis Based on Abstract Interpretation. In *Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8702)*. Springer, 271–277. https://doi.org/10.1007/978-3-319-10431-7_20

[10] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

[11] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski (Eds.). ACM Press, 84–96. https://doi.org/10.1145/512760.512770

[12] Pietro Ferrara, Luca Negrini, Vincenzo Arceri, and Agostino Cortesi. 2021. Static analysis for dummies: experiencing LiSA. In *SOAP@PLDI 2021: Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, Virtual Event, Canada, 22 June, 2021*, Lisa Nguyen Quang Do and Caterina Urban (Eds.). ACM, 1–6. https://doi.org/10.1145/3460946.3464316

[13] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2021. Disjunctive Interval Analysis. In *Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12913)*, Cezara Dragoi, Suvam Mukherjee, and Kedar S. Namjoshi (Eds.). Springer, 144–165. https://doi.org/10.1007/978-3-030-88806-0_7

[14] Arie Gurfinkel and Sagar Chaki. 2010. Boxes: A Symbolic Abstract Domain of Boxes. In *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6337)*, Radhia Cousot and Matthieu Martel (Eds.). Springer, 287–303. https://doi.org/10.1007/978-3-642-15769-1_18

[15] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9206)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 343–361. https://doi.org/10.1007/978-3-319-21690-4_20

[16] Arie Gurfinkel and Jorge A. Navas. 2021. Abstract Interpretation of LLVM with a Region-Based Memory Model. In *Software Verification - 13th International Conference, VSTTE 2021, New Haven, CT, USA, October 18-19, 2021, and 14th International Workshop, NSV 2021, Los Angeles, CA, USA, July 18-19, 2021, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 13124)*, Roderick Bloem, Rayna Dimitrova, Chuchu Fan, and Natasha Sharygina (Eds.). Springer, 122–144. https://doi.org/10.1007/978-3-030-95561-8_8

[17] Julien Henry, David Monniaux, and Matthieu Moy. 2012. PAGAI: A Path Sensitive Static Analyser. In *Third Workshop on Tools for Automatic Program Analysis, TAPAS 2012, Deauville, France, September 14, 2012 (Electronic Notes in Theoretical Computer Science, Vol. 289)*. Elsevier, 15–25. https://doi.org/10.1016/j.entcs.2012.11.003

[18] Bertrand Jeannet and Antoine Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5643)*, Ahmed Bouajjani and Oded Maler (Eds.). Springer, 661–667. https://doi.org/10.1007/978-3-642-02658-4_52

[19] Sung Kook Kim, Arnaud J. Venet, and Aditya V. Thakur. 2020. Memory-Efficient Fixpoint Computation. In *Static Analysis - 27th International Symposium, SAS 2020, Virtual Event, November 18-20, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12389)*, David Pichardie and Mihaela Sighireanu (Eds.). Springer, 35–64. https://doi.org/10.1007/978-3-030-65474-0_3

[20] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2021. A Multilanguage Static Analysis of Python Programs with Native C Extensions. In *Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12913)*, Cezara Dragoi, Suvam Mukherjee, and Kedar S. Namjoshi (Eds.). Springer, 323–345. https://doi.org/10.1007/978-3-030-88806-0_16

[21] Luca Negrini, Vincenzo Arceri, Pietro Ferrara, and Agostino Cortesi. 2021. Twinning Automata and Regular Expressions for String Static Analysis. In *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12597)*, Fritz Henglein, Sharon Shoham, and Yakir Vizel (Eds.). Springer, 267–290. https://doi.org/10.1007/978-3-030-67067-2_13

[22] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2017. Fast polyhedra abstract domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 46–59. https://doi.org/10.1145/3009837.3009885

Received 2023-03-10; accepted 2023-04-21

When Long Jumps Fall Short

Control-Flow Tracking and Misuse Detection for Non-local Jumps in C

Michael Schwarz
Technische Universität München
Garching, Germany
m.schwarz@tum.de

Julian Erhard
Technische Universität München
Garching, Germany
julian.erhard@tum.de

Vesal Vojdani
University of Tartu
Tartu, Estonia
vesal.vojdani@ut.ee

Simmo Saan
University of Tartu
Tartu, Estonia
simmo.saan@ut.ee

Helmut Seidl
Technische Universität München
Garching, Germany
helmut.seidl@tum.de

Abstract

The C programming language offers `setjmp/longjmp` as a mechanism for non-local control flow. This mechanism has complicated semantics. As most developers do not encounter it day-to-day, they may be unfamiliar with all its intricacies – leading to subtle programming errors. At the same time, most static analyzers lack proper support, implying that otherwise sound tools miss whole classes of program deficiencies. We propose an approach for lifting existing interprocedural analyses to support `setjmp/longjmp`, as well as to flag their misuse. To deal with the non-local semantics, our approach leverages side-effecting transfer functions which, when executed, may trigger contributions to extra program points. We showcase our analysis on real-world examples and propose a set of litmus tests for other analyzers.

CCS Concepts: • **Theory of computation** → **Program analysis**; • **Software and its engineering** → *Software verification and validation.*

Keywords: Abstract Interpretation, Static Analysis, setjmp / longjmp, Side-Effects

ACM Reference Format:
Michael Schwarz, Julian Erhard, Vesal Vojdani, Simmo Saan, and Helmut Seidl. 2023. When Long Jumps Fall Short: Control-Flow Tracking and Misuse Detection for Non-local Jumps in C. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '23), June 17, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3589250.3596140

SOAP '23, June 17, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0170-2/23/06.
https://doi.org/10.1145/3589250.3596140

1 Introduction

For statically analyzing real-world programs, analysis developers are confronted with a wealth of intricate language features. Therefore, analyzers often focus on a subset of the programming language, ignoring some more obscure features and making optimistic assumptions about others. As noted by Livshits et al. [12], this is true even for tools that claim to be *sound*, i.e., not to miss any bug. The authors provide a checklist for static analyzers, making it easier for analysis authors to indicate which features are supported and which are not. For the C programming language, they mention `setjmp/longjmp` as an example of often unsupported language features. The functions `setjmp/longjmp` allow defining exceptional control-flow by dynamically jumping up the callstack. Indeed, these non-local jumps are frequently ignored [2, 3, 10, 11, 18]. Empirical studies by Christakis and Bird [5] indicate that developers think it of exceptional importance to cover exceptional control flow. Therefore, the lack of support for this C language feature is unsatisfactory.

The `setjmp/longjmp` mechanism allows saving the current state of execution into a *jumpbuffer* by means of `setjmp`. At a later point of program execution, parts of the callstack may be abandoned by calling `longjmp`. Execution then continues at the stackframe specified by the jumpbuffer. This is not only conceptually intricate, but fraught with many caveats. For example, accessing the values of non-volatile locals that have been modified between the call to `setjmp` and the call to `longjmp` is Undefined Behavior. As developers typically only use `setjmp/longjmp` in few select locations, familiarity with these intricacies is not widely spread, resulting in potential vulnerabilities [13, 14]. Therefore, a static analyzer should not ignore `setjmp/longjmp`. Instead, not to fall short of user expectations, it should take this feature into account during its analysis of other programming deficiencies, *as well as* warn about potential misuses. While exception handlers in Java or C++ are well-structured, this is not necessarily the case for `setjmp/longjmp` where the jump target may depend on the jumpbuffer's runtime value, making the analysis of `setjmp/longjmp` challenging.

```
1  jmp_buf_t errorhandler;
2  int err;
3  void foo() {
4    // ...
5    int z = get_status();
6    if(z < 0) {
7      err = 42;
8      longjmp(errorhandler, err);
9    }
10   // ...
11 }
12 void bar() {
13   char* logpath;
14   if(setjmp(errorhandler)) {
15     printf("error encountered!");
16     // Bug!
17     write_log(logpath, err);
18     exit(1);
19   }
20   logpath = get_logpath();
21   foo();
22 }
23 void main() {
24   bar();
25   // ...
26   int z = get_status();
27   if(z < 0) {
28     err = -17;
29     // Bug!
30     longjmp(errorhandler, err);
31   }
32 }
```

Figure 1. Program fragment making use of `setjmp/longjmp` that contains two bugs.

We took up this challenge, and demonstrate how existing analyses for C can be lifted to support `setjmp/longjmp` by reusing existing building blocks of interprocedural analyses, instead of dedicated mechanisms. We propose the analysis perform an *abstract stack unwinding* complemented with

- an analysis of currently valid jump targets;
- a value analysis for jump buffers, and
- taint analyses to check for illegal accesses to locals.

For stack unwinding and the collection of abstract states at `setjmp` locations, we rely on *side-effects* in transfer functions. Side-effecting constraint systems [1, 21] allow accumulating flow-insensitive information during a flow- and context-sensitive analysis. They have been used, e.g., for the analysis of the values of global variables [19, 20] or the accesses to globals to check for races [23]. Here, side-effects are used to handle `longjmps` without polluting control-flow graphs with an excessive number of additional edges (e.g., from every procedure call to every invocation of `setjmp`).

The rest of the paper is structured as follows. In Section 2, we recall the semantics of `setjmp/longjmp` along an example and identify possible programming errors. Section 3 recalls side-effecting constraint systems and indicates our approach to inter-procedural analysis. Section 4 presents the analysis of `setjmp/longjmp`, while Section 5 explains how illegal accesses to locals are identified. Section 6 reports on our implementation within an analyzer for multi-threaded C based on Abstract Interpretation [6] and the results of a preliminary experimental evaluation. Finally, Section 7 discusses related work whereas Section 8 concludes.

2 Setjmp/Longjmp in C

The usage of `setjmp/longjmp` is best explained by an example. The program in Fig. 1 has two global variables, namely, `errorhandler` of the pre-defined type `jmp_buf` and `err` of type `int`. The `main` function first calls `bar` that in turn calls `setjmp(errorhandler)`. That call saves the current state of execution into the jumpbuffer `errorhandler`, returning the value `0`. Thus, the `if` branch in line 14 is not taken. Function `bar` then sets its local variable `logpath` to the path returned by a call to the external function `get_logpath()`. The program is meant to later record error messages into the file identified by `logpath`. The function `bar` continues with calling `foo`. Inside `foo` at line 5, the status information as returned by some function `get_status()`, is checked. A negative status is interpreted as an error indication, in which case `err` is set to 42 and `longjmp(errorhandler, err)` is invoked. This call transfers control back to the invocation of `setjmp` inside of `bar` at line 14 – which now returns with the value 42 passed as the argument to `longjmp`. Now, the `if` branch is taken, a message is printed to `stdout`, an error message written into the file, and the program terminates.

If no error occurs during `foo` on the other hand, `foo` and `bar` both return regularly. Subsequently, the function `main` checks the status again, and, if it is negative, `err` is set to `-17` and a call to `longjmp(errorhandler, err)` occurs. This, though, constitutes a fault in the program, as it performs a `longjmp` to an invocation of `setjmp` inside a function that already has returned, resulting in Undefined Behavior.

However, this is not the only fault in the fragment: Consider the case where a `longjmp` happens from `foo` at line 8. Here, the function `bar` containing the call to `setjmp` has *not* returned yet. The issue is different: the local variable `logpath` has been modified between the call to `setjmp` and the call to `longjmp`. Thus, it has Indeterminate Value after the `longjmp`, resulting in Undefined Behavior at the access in line 17. There is an easy fix though: Declaring `logpath` to be `volatile` ensures that the correct value is read.

The example highlights pitfalls of using `setjmp/longjmp`, which a static analyzer should be expected to flag. The following misuses are possible and lead to Undefined Behavior:

(A) Calling `longjmp` on a `jmp_buf` for which `setjmp` has not been called.
(B) Calling `longjmp` when the function containing the corresponding call to `setjmp` has already returned.
(C) Calling `longjmp` from a different thread than the one calling `setjmp`.
(D) Reading a non-volatile variable x of automatic storage duration after a `longjmp` where x has been modified between `setjmp` and `longjmp` and has not been overwritten since the `longjmp`.

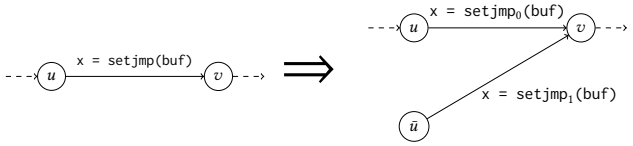


Figure 2. Transformation of the CFG introducing an extra edge and node for calls to `setjmp`.

(E) Calling `longjmp` on a `jmp_buf` that has not been initialized via `setjmp` but instead by copying the content of a different `jmp_buf`.
(F) Calling `longjmp` where the corresponding `setjmp` was within the scope of a variable-length array and this scope has since been left.

Additionally, (G) if argument 0 is passed to `longjmp` this argument is silently changed to 1. While this is not Undefined Behavior, it still likely is a bug and a warning should be produced. Furthermore, (H) any code after a call to `longjmp` is unreachable, for which an appropriate warning should also be issued. We will discuss how to soundly detect all these possible misuses and issue warnings in Section 4.

3 Preliminaries

We rely on *side-effecting constraint systems* as applied to interprocedural analysis in [1]. Here, we indicate how side-effecting constraint systems conveniently allow handling `longjmps`: Side-effects are used both to collect the abstract states with which one may return to the `setjmp` location, and to propagate these states to this location. Besides avoiding polluting the control-flow graphs (CFGs) with an excessive number of additional edges, this allows handling *dynamic* calls to both `setjmp` and `longjmp` via pointers.[1]

Consider a side-effecting constraint system over a set of unknowns $\mathcal{X}$ taking values from some complete lattice $\mathbb{D}$. The right-hand side f_x for an unknown x is of the form $(\mathcal{X} \Rightarrow \mathbb{D}) \to ((\mathcal{X} \Rightarrow \mathbb{D}) \times \mathbb{D})$ where $\mathcal{X} \Rightarrow \mathbb{D}$ denotes all mappings from unknowns to abstract values. Thus, f_x takes a mapping and returns as its first component a mapping collecting the contributions to *other* unknowns (the *side-effects*) and as the second component the contribution to the left-hand side x. We call a mapping η a *solution* of the system, if for all unknowns $x \in \mathcal{X} : f_x\, \eta \sqsubseteq (\eta, \eta\, x)$ where $\sqsubseteq$ is lifted point-wise to maps and tuples. Such solutions can, e.g., be computed by local solvers [22].

A program is a set of CFGs, each labelled with the name of the respective function, one of which is `main`. The nodes of the CFGs are from a finite set $\mathcal{N}$, with st_f and ret_f referring to the (unique) start and end points of function f. For simplicity, we assume that the return value of the function, if any, is assigned to a dedicated program variable retv.

[1]Even though `setjmp`, according to the standard, may not be permitted as part of an assignment, e.g., the GCC implementation allows this.

For context-sensitive analysis, the *unknowns* are pairs of program points and *contexts* α, which we denote by $[u, \alpha]$. Let $\mathbb{C}$ be some set of contexts. Analysis is relative to some initial context in which `main` is analyzed. We only give the right-hand sides for function calls here. A control-flow edge $e = (u, x = f(a_1, \ldots, a_n), v)$ and context $\alpha \in \mathbb{C}$ give rise to the constraint $(\eta, \eta\,[v, \alpha]) \sqsupseteq [\![e, \alpha]\!]^\sharp\ \eta$ where the right-hand side $[\![e, \alpha]\!]^\sharp\ \eta$ is assembled from helper functions $\mathrm{enter}_e^\sharp : \mathbb{D} \to (\mathbb{C} \times \mathbb{D})$ and $\mathrm{combine}_x^\sharp : \mathbb{D} \to \mathbb{D} \to \mathbb{D}$ as follows:[2]

$$
\begin{aligned}
[\![e, \alpha]\!]^\sharp\ \eta = {} & \textbf{let}\ \sigma\ = \eta\,[u, \alpha]\ \textbf{in} \\
& \textbf{let}\ \langle \beta, \sigma' \rangle = \mathrm{enter}_e^\sharp\ \sigma\ \textbf{in} \\
& \textbf{let}\ \sigma'' = \mathrm{combine}_x^\sharp\ \sigma\ (\eta\,[\mathrm{ret}_f, \beta])\ \textbf{in} \\
& (\{[\mathrm{st}_f, \beta] \mapsto \sigma'\}, \sigma'')
\end{aligned}
$$

Here, $\mathrm{enter}_e^\sharp$ takes the local state of the caller and yields a pair. Its second component is an entry state for the function f obtained by, e.g., removing unreachable locals of the caller and assigning (abstract) values for the actuals to the formals. The first component β is the calling context for f. The entry state is then side-effected to the unknown $[\mathrm{st}_f, \beta]$ corresponding to the entry node of f in context β.

The function $\mathrm{combine}_x^\sharp$ takes as its first argument the local state of the caller and as its second argument the state computed for the endpoint of function f in context β, i.e., the value of unknown $[\mathrm{ret}_f, \beta]$, and combines these states into the state after the function call. This entails, e.g., removing local variables of f and assigning the value of `retv` to the variable x on the left-hand side of the call.

For our analysis of programs with `setjmp/longjmp`, we enhance the base approach by extending the abstract domain $\mathbb{D}$ to express auxiliary information and lifting all right-hand sides accordingly. New transfer functions corresponding to `setjmp` and `longjmp` are assembled from the abstract effects of assignments, $\mathrm{enter}_e^\sharp$, and $\mathrm{combine}_x^\sharp$.

4 Analysis of `Setjmp/Longjmp`

As a first step, we transform the CFG to facilitate analysis (see Fig. 2). According to the C standard, it appears as if a call to `setjmp` may return more than once: First when `setjmp` is invoked, and possibly again whenever a `longjmp` back to this program point occurred. We make this explicit by replacing each edge $(u, \texttt{x=setjmp(buf)}, v)$ with edges $(u, \texttt{x=setjmp}_0\texttt{(buf)}, v)$ and $(\bar{u}, \texttt{x=setjmp}_1\texttt{(buf)}, v)$ where unknowns associated with $\bar{u}$ receive their values by side-effects only. The first edge corresponds to `setjmp` returning for the first time, while the second edge corresponds to all further returns, i.e., `longjmps` to this location. Let $\mathcal{B}$ denote the set of the newly introduced barred nodes $\bar{u}$.

For analyzing `setjmp/longjmp`, we identify three tasks:

[2]Throughout this paper, we assume there are only function calls with statically known targets. Our implementation also deals with calls via pointers. We also assume that each function call constitutes a separate statement.

- **P1** Tracking which targets may legally be jumped to, i.e., keeping track of all invocations of `setjmp` where the containing call has not returned yet.
- **P2** Tracking the values of variables of type `jmp_buf`.
- **P3** Using information from **P1** and **P2**, propagating states from `longjmp` to `setjmp`.

In order to realize items **P1** and **P2** we first introduce an abstract domain $\mathbb{D}_{\mathsf{buf}}$ to track values of variables of type `jmp_buf`. The domain consists of sets of pairs of barred nodes and contexts, i.e, $\mathbb{D}_{\mathsf{buf}} = 2^{\mathcal{B}\times\mathbb{C}}$ ordered by $\subseteq$.

For **P1**, we enhance the base analysis to path- and context-sensitively track the set of legal jumptargets in an auxiliary local variable legal which receives values from $\mathbb{D}_{\mathsf{buf}}$. A call to `setjmp` adds the pair $\langle \bar{u}, \alpha\rangle$ for current node u and calling context α to the value of legal. The set of legal jumptargets is passed to callees, but upon combine, the set from the caller is restored, as the callee has already returned: any jumptargets inside the callee are no longer legal.

For realizing **P2**, the base analysis is enhanced along the lines of an off-the-shelf *values-of-variables* analysis to track for each variable of type `jmp_buf` a value from $\mathbb{D}_{\mathsf{buf}}$, extended with an error value for representing *invalid* values. *Invalid* here means values not assigned by `setjmp`. The transfer function for `x=setjmp`$_0$`(b)` applied to node u in context α sets the value for `b` to the set $\{\langle \bar{u}, \alpha\rangle\}$ and sets `x` to `0`.

Now, we can give the right-hand sides of the analysis to accomplish **P3**. For function calls, the single return node ret_f is complemented with a secondary node ret'_f where function f is exited via a `longjmp` in f itself or in transitively called functions. The values at unknowns for ret'_f consist of sets of pairs of current jumptarget and abstract state.

`Longjmp`s. Assume that $e = (u, \texttt{longjmp(b,a)}, v)$. Then

$$
\begin{aligned}
&[\![e, \alpha]\!]^\sharp\ \eta = \textbf{let}\ \sigma = \eta\,[u, \alpha]\ \textbf{in}\\
&\ \textbf{let}\ T = [\![b]\!]^\sharp\ \sigma\ \textbf{in}\\
&\ \textbf{let}\ L = T \cap (\sigma\ \mathsf{legal})\ \textbf{in}\\
&\ \textbf{let}\ \_ = \textbf{if}\ T \setminus L \neq \emptyset\ \textbf{then warn}\ \text{"illegal targets"}\ \textbf{in}\\
&\ \textbf{let}\ \sigma' = [\![\texttt{retv = (a==0 ? 1 : a)}]\!]^\sharp\ \sigma\ \textbf{in}\\
&\ \textbf{let}\ \rho_{\mathrm{h}} = \{[\bar{u}_j, \alpha] \mapsto \sigma' \mid \langle \bar{u}_j, \gamma\rangle \in L \wedge \mathsf{here}_{u,\alpha}\langle \bar{u}_j, \gamma\rangle\}\ \textbf{in}\\
&\ \textbf{let}\ \rho_{\mathrm{o}} = \{[\mathsf{ret}'_{\mathsf{func}\,u}, \alpha] \mapsto \{(t, \sigma') \mid t \in L \wedge \neg\mathsf{here}_{u,\alpha}\, t\}\ \textbf{in}\\
&\ (\rho_{\mathrm{h}} \cup \rho_{\mathrm{o}}, \bot)
\end{aligned}
$$

where $\mathsf{func}\,u$ takes a program point u, and returns the function u is in, and $\mathsf{here}_{u,\alpha}\langle \bar{u}_j, \gamma\rangle := (\mathsf{func}\,u = \mathsf{func}\,\bar{u}_j) \wedge (\alpha = \gamma)$ checks if its argument is a jumptarget that is local to the current function and calling context.

Thus, the right-hand side for a `longjmp` first evaluates the expression b to a set of jumptargets, and warns if any of these jumptargets is not legal. Then the return state σ' is prepared. The dedicated variable retv is used to receive the return values of function calls as well as the values to be returned by the call to `setjmp`. Care is taken to set the value of `retv` to `1` should `a` be `0`. If the analysis cannot exclude `0` for `a`, an appropriate warning is issued (omitted for clarity). The assignment with a ternary expression may be broken down into guards and plain assignments by the analysis. Then, for those jumptargets $\langle \bar{u}_j, \gamma\rangle$ *within* the given function and abstract calling context, a side-effect to the unknown $[\bar{u}_j, \gamma]$ for arriving via `longjmp` is produced. For jumptargets *outside* the current function and abstract calling context, a side-effect to the unknown $[\mathsf{ret}'_{\mathsf{func}\,u}, \alpha]$ for returning from the current function via `longjmp` is produced. The contribution to the control-flow successor of the `longjmp` statement is $\bot$, as the call to `longjmp` does not return.

Function Calls. For the right-hand sides corresponding to function calls, now on top of accounting for a normal return in the manner introduced in Section 3, potentially occurring `longjmp`s must be handled. These may occur in the function itself or within some nested call. Again, a case distinction is required on whether the jumptarget occurs within the current function and context or not. Let $e = (u, x = f(a_1, \ldots, a_n), v)$ with $\mathsf{func}\,u = g$.

$$
\begin{aligned}
&[\![e, \alpha]\!]^\sharp\ \eta = \textbf{let}\ \sigma\ = \eta\,[u, \alpha]\ \textbf{in}\\
&\ \textbf{let}\ \langle \beta, \sigma'\rangle = \mathsf{enter}^\sharp_e\ \sigma\ \textbf{in}\\
&\ \textbf{let}\ \sigma'' = \mathsf{combine}^\sharp_x\ \sigma\ (\eta\,[\mathsf{ret}_f, \beta])\ \textbf{in}\\
&\ \textbf{let}\ J = \{(t, \mathsf{combine}^\sharp_{\mathsf{retv}}\ \sigma\ \sigma_j) \mid (t, \sigma_j) \in \eta\,[\mathsf{ret}'_f, \beta]\}\ \textbf{in}\\
&\ \textbf{let}\ \rho_{\mathrm{h}} = \{[\bar{u}_j, \alpha] \mapsto \sigma_j \mid (\langle \bar{u}_j, \gamma\rangle, \sigma_j) \in J \wedge \mathsf{here}_{u,\alpha}\langle \bar{u}_j, \gamma\rangle\}\textbf{in}\\
&\ \textbf{let}\ \rho_{\mathrm{o}} = \{[\mathsf{ret}'_g, \alpha] \mapsto \{(t, \sigma_j) \mid (t, \sigma_j) \in J \wedge \neg\mathsf{here}_{u,\alpha}\, t\}\}\ \textbf{in}\\
&\ (\{[\mathsf{st}_f, \beta] \mapsto \sigma'\} \cup \rho_{\mathrm{h}} \cup \rho_{\mathrm{o}}, \sigma'')
\end{aligned}
$$

The right-hand side now accesses the unknown $[\mathsf{ret}'_f, \beta]$ accounting for leaving f via `longjmp`. The value of this unknown is a set of pairs of jumptargets and local states. Recall that in these local states σ_j, the value supplied as the second argument to the call of `longjmp` has been written to the variable `retv`. Each local state σ_j is combined with the local state σ, to account for all modifications the current function made before calling f (including modification of locals of other functions to which a pointer was passed), and to discard the locals of f. To preserve the value of `retv`, the combine mechanism is used for `retv` instead of `x`. The resulting set of pairs of jumptargets and local states of the current function g is collected in J. Those local states for jumptargets $\langle \bar{u}_j, \alpha\rangle$ in J which are inside the current function and context, are side-effected to the unknowns $[\bar{u}_j, \alpha]$. The pairs of targets and values from J for *other* jumptargets are propagated to the dedicated unknown $[\mathsf{ret}'_g, \alpha]$ for `longjmp`ing out of g.

`Setjmp`s. The right-hand side for `setjmp`$_1$ in calling context α takes its argument from some unknown $[\bar{u}, \alpha]$ where all states from arriving `longjmp`s have been collected via side-effects. Recall that the return value of `longjmp` is tracked via the return variable `retv`. `setjmp`$_1$ then behaves as the assignment `x=retv`.

The analysis thus performs a *stack-unwinding* that is not performed in the concrete, to account for all effects of function calls further up the stack before termination via `longjmp`.

Notably, this includes modifications to local variables that have escaped their function. Performing this stack-unwinding in the abstract does not cause any further imprecision.

Example 4.1. Assuming that the context for the call to `bar` in `main` in Fig. 1 is $\alpha_{\texttt{bar}}$, the set of legal jumptargets is set to $L = \{\langle \bar{u}_{14}, \alpha_{\texttt{bar}}\rangle\}$ by the `setjmp` in line 14. L is passed to the call to `foo` in context $\alpha_{\texttt{foo}}$. At the call of `longjmp` in line 8, the abstract value for `errorhandler` is $\{\langle \bar{u}_{14}, \alpha_{\texttt{bar}}\rangle\}$. As it contains only elements in L, all jumptargets are legal. The analysis then assigns the abstract value for `err` to `retv`, resulting in some abstract state σ'. As the jumptarget is not inside the current function, the value $\{p\}$ with $p = (\langle \bar{u}_{14}, \alpha_{\texttt{bar}}\rangle, \sigma')$ is side-effected to the special return unknown $[\text{ret}'_{\texttt{foo}}, \alpha_{\texttt{foo}}]$, and any code after the `longjmp` is marked as dead (although, here, there is none). Now consider the call to `foo` in line 21 of function `bar`. For leaving `foo` via `longjmp`, the value of $[\text{ret}'_{\texttt{foo}}, \alpha_{\texttt{foo}}]$ is accessed. As the jumptarget in p is in the current function and context, the state σ'' obtained by combining σ' with the local state at line 21 is side-effected to unknown $[\bar{u}_{14}, \alpha_{\texttt{bar}}]$. This state, modified by the effect of the assignment `x = retv`, accounts for returning to line 14 via `longjmp` from `foo`. For the call to `longjmp` in line 30, on the other hand, the jumptarget is illegal, and an appropriate warning is produced. □

This analysis handles correct usages of `setjmp/longjmp`, detects dead code following a call to `longjmp` **(H)**, warns if the second argument to `longjmp` may be `0` **(G)**, and detects illegal invocations of `longjmp` **(A, B)**. To detect when `longjmp` is called from a different thread than `setjmp` **(C)**, an analysis of thread *id*s is required as, e.g., provided in [20].

To additionally achieve **(E)**, the *values-of-variables* analysis should be extended to track whether the value of a variable of type `jmp_buf` was set via an invocation of `setjmp` or not. To achieve **(F)**, it suffices to collect for each `setjmp`, the set of scopes defining variably sized objects, and then track for each `longjmp` as well as each call possibly terminated by means of a `longjmp`, the set of such scopes that have potentially been left. Our current implementation does not perform this detailed analysis, but instead warns when `setjmp`s happen in scopes defining variably sized objects.

5 Indeterminate Local Variables

It remains to detect whether non-volatile local variables have been modified since the call to `setjmp` **(D)**. We take a three-pronged approach by combing three different taint analyses (T_{intra}, T_{inter}, and T_{poison}). Inside the function that performs the call to `setjmp`, T_{intra} tracks the set of potentially modified non-volatile locals of automatic storage duration jumptarget-sensitively. For a called function f, T_{inter} tracks the set of those passed to and potentially modified by f – independently of jumptargets. This is justified since few local variables are usually passed to called functions. T_{inter} employs an interprocedural taint-analysis, which we do not detail here. Lastly, T_{poison} tracks variables that have Indeterminate Value after a `longjmp` (these variables are said to be *poisonous*), and warns upon access to such variables.

Let $\mathcal{V}_{\texttt{f}}$ the set of *non-volatile* local variables of automatic storage duration occurring in a function `f`. For the *intraprocedural* jumptarget-sensitive analysis T_{intra}, we employ an extra domain $\mathbb{D}_{\text{taint}} = \mathcal{B} \to 2^{\mathcal{V}_{\texttt{f}}}$ consisting of mappings from barred nodes to sets of local variables that potentially have been written since the corresponding jumptarget was set. The ordering on $\mathbb{D}_{\text{taint}}$ is given by $\subseteq$ lifted to partial maps, i.e., $\mu_1 \subseteq \mu_2$ iff $\mu_1\, a \subseteq \mu_2\, a$ whenever $\mu_1\, a$ is defined. Functions are entered with the empty partial mapping. When a `setjmp` is encountered at $[u, \alpha]$, the binding $\bar{u} \mapsto \emptyset$ is added to the mapping. Whenever a relevant local is modified, it is added to *each* set in the mapping. For each function call, T_{inter} yields a set of relevant locals possibly modified during the call, which is once again added to each set in the mapping.

T_{poison} maintains for every program point and context, a set of possibly poisonous variables P. Consider a `longjmp` at program point u in context α, and assume that T_{intra} has determined for $[u, \alpha]$ the mapping μ. For every barred node $\bar{u}_j$ within the current function where μ is defined, P additionally receives the set $\mu\, \bar{u}_j$. Function calls are treated similarly. Starting from the jumptargets, the set P of possibly poisonous variables is propagated. Whenever a variable in P is *definitely* assigned to, it is removed from P, as the indeterminate value is overwritten. If a variable in P *may* be accessed, the analyzer warns that an indeterminate value may be read.

Example 5.1. After the `setjmp` in line 14 of `bar` in Fig. 1, the map from barred nodes to potentially written local variables is set to $\{\bar{u}_{14} \mapsto \emptyset\}$. In line 20, the non-volatile local variable `logpath` is modified, resulting in $\{\bar{u}_{14} \mapsto \{\texttt{logpath}\}\}$. After `foo` is left via `longjmp`, the set of poisonous variables is $\{\texttt{logpath}\}$. Since `logpath` is not overwritten before being accessed on line 17, a warning is produced. □

We have thus enhanced the analysis with taint analyses to not only remain sound in the presence of `setjmp/longjmp`, but also flag all possible issues (**A**) - (**H**), thus enabling developers to use `setjmp/longjmp` without fear of falling short.

6 Implementation and Evaluation

We have implemented the analysis within the analysis framework Goblint[3] for multi-threaded C. We leveraged existing domains, and the support for *path-sensitivity* (in the return values of calls to `setjmp`). No changes to solvers were necessary. The implementation performs all analyses jointly. We evaluated it in two ways: First, we extracted challenging usages of `setjmp/longjmp` from real-world programs and crafted 45 litmus tests for semantic issues of `setjmp/longjmp` and verified that the analyzer passes these. These may serve as sanity checks for future analyzers.

[3] https://goblint.in.tum.de and https://github.com/goblint/analyzer

The `libpng`[4] is a C library heavily using `setjmp/longjmp`. Several bugs in uses of this library have been reported[5]. To check real-world applicability, we analyzed the `pngtest` program[6] that comes with that library. The number of warnings for `setjmp/longjmp` related issues is reasonable: Two warnings about accesses to a single distinct poisonous variables are reported, as well as one warning about jumping to an unknown jumpbuffer, one warning about jumping to a jumpbuffer with a value not set by `setjmp`, and two warnings about jumping to potentially invalid targets. Manual inspection reveals most of the warnings to be spurious: Goblint's loss of precision here is due to `libpng`'s heavy usage of dynamically allocated memory objects with structs containing jumpbuffers, pointers to jumpbuffers etc. To get rid of the spurious warnings, a more sophisticated heap analysis would be required. The warnings about accesses to the poisonous variable `row_buf`, however, seem to be indicative of Undefined Behavior. We could not observe any misbehavior of the binary compiled with Gcc – perhaps, because the address of this pointer variable escapes to an external function preventing assignment to a register. A programmer, however, should not rely on compiler implementation details. An analogous usage pattern has, e.g., resulted in a memory leak in an older version of ImageMagick [16]. That bug when injected into `pngtest`, is correctly detected by Goblint. For a less heap-intensive benchmark, we considered the exceptions library proposed by Roberts [17]. The use of the library can result in Undefined Behavior if non-volatile variables are updated between the *try*-clause and the *catch*-clause as these macros are translated into `setjmp/longjmp`.[7] Goblint detects cases when the client code elicits Undefined Behavior, while Gcc 12 and Clang 14 (with `-O2`) clobber such variables – without issuing warnings. We also evaluated the performance penalty of enabling this analysis on the SV-COMP 2022 benchmarks and observed a slowdown of 17%. This can be remedied though by a pre-analysis that only enables this analysis for programs where `longjmp` is statically called or referenced via pointer. Our implementation as well as the litmus tests and the larger programs are publicly available.[8]

7 Related Work

Feng et al. [7] give a Hoare-style framework for the verification of assembly code including support for stack-based control abstractions such as `setjmp/longjmp`. We aim for *automatic* techniques. Non-local control flow has also been considered for an analysis of Python code [8] (elaborated on in [15]), and in particular to handle generators. They do not rely on side-effecting constraint systems. Instead, they iterate over the syntax and account for non-local flow which, unlike in our setting, also includes `break` and `continue` in loops by partitioning according to *flow tokens*. States associated with some of these tokens are incorporated into the current flow when the respective program point is encountered. Like in our setting, targets in which execution is to be resumed need to be tracked. The challenges, though, are different as control-flow for generators is more structured: Upon a call to `next`, execution resumes at the beginning of the generator or its last call to `yield`. Upon calling `yield` in the generator, control is returned to the caller of `next`, and execution continues at that program point. It is not immediately obvious how to craft iteration over the syntax for dealing with `setjmp/longjmp`: A call to `longjmp` may transfer control to program points that are not static control-flow successors of any of the calls containing `longjmp`s. The analysis of exceptions in languages such as Java or C++ is also closely related: see, e.g., [4] for a recent a comprehensive survey. As our approach intertwines the analysis of non-local control flow with other analyses, e.g., of points-to information, it is most closely related to the class of *Combined Exceptional Control Flow Analyses* identified in the survey. However, analyzing `setjmp/longjmp` poses challenges not faced when analyzing C++ or Java programs where non-local control flow is well-structured. Wilson [24] provides an account of handling `setjmp/longjmp` in a context-sensitive summary-based pointer analysis: His approach introduces a second function summary accounting for leaving the function via `longjmp`. Unlike our approach, he does not track values of jumpbuffers and assumes that a `longjmp` may jump to any `setjmp` further up the callstack. The analysis only endeavors to compute sound pointer information in the presence of correct usages of `setjmp`, and does not detect possible misuses of the feature, as ours does. [9] claims to use a similar technique to Wilson, but does not elaborate on details.

8 Conclusions and Future Work

We have provided a novel technique for lifting static analyses to support `setjmp/longjmp`. Dynamic control-flow is dealt with by side-effects in transfer functions. We discussed how to analyze programs containing correct usages of `setjmp/longjmp`, and how to detect potentially incorrect usages. We have enhanced the static analyzer Goblint with the technique and evaluated it on challenging litmus tests as well as on real-world programs. In the future, tackling further "dark corners" of the C standard that have also not seen widespread support from static analyzers, may provide new and interesting challenges for crafting analyzers capturing more real-world aspects of programming.

Acknowledgments. We thank Benjamin Bott for implementing an earlier prototype. This work was supported by Deutsche Forschungsgemeinschaft (DFG) – 378803395/2428 ConVeY and the Estonian Centre of Excellence in IT, funded by the European Regional Development Fund.

[4] http://www.libpng.org/pub/png/libpng.html
[5] e.g. in ImageMagick commits 75fc6 and e88c1. See also [16]
[6] Around 7000 logical lines of live code
[7] https://github.com/cs50/spl/issues/24
[8] https://github.com/goblint/bench/tree/longjmp/setjmp

References

[1] Kalmer Apinis, Helmut Seidl, and Vesal Vojdani. 2012. Side-Effecting Constraint Systems: A Swiss Army Knife for Program Analysis. In *Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7705)*, Ranjit Jhala and Atsushi Igarashi (Eds.). Springer, 157–172. https://doi.org/10.1007/978-3-642-35182-2_12

[2] Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. 2015. Static Analysis and Verification of Aerospace Software by Abstract Interpretation. *Found. Trends Program. Lang.* 2, 2-3 (2015), 71–190. https://doi.org/10.1561/2500000002

[3] Sandrine Blazy, David Bühler, and Boris Yakobowski. 2017. Structuring Abstract Interpreters Through State and Value Abstractions. In *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10145)*, Ahmed Bouajjani and David Monniaux (Eds.). Springer, 112–130. https://doi.org/10.1007/978-3-319-52234-0_7

[4] Byeong-Mo Chang and Kwanghoon Choi. 2016. A review on exception analysis. *Inf. Softw. Technol.* 77 (2016), 1–16. https://doi.org/10.1016/j.infsof.2016.05.003

[5] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 332–343. https://doi.org/10.1145/2970276.2970347

[6] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

[7] Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni. 2006. Modular verification of assembly code with stack-based control abstractions. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM, 401–414. https://doi.org/10.1145/1133981.1134028

[8] Aymeric Fromherz, Abdelraouf Ouadjaout, and Antoine Miné. 2018. Static Value Analysis of Python Programs by Abstract Interpretation. In *NASA Formal Methods - 10th International Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10811)*, Aaron Dutle, César A. Muñoz, and Anthony Narkawicz (Eds.). Springer, 185–202. https://doi.org/10.1007/978-3-319-77935-5_14

[9] Michael Hind and Anthony Pioli. 2001. Evaluating the effectiveness of pointer alias analyses. *Sci. Comput. Program.* 39, 1 (2001), 31–55. https://doi.org/10.1016/S0167-6423(00)00014-9

[10] Xavier Leroy. 2022. *The CompCert C verified compiler - Documentation and user's manual - Version 3.12.* Technical Report. Collège de France and INRIA.

[11] Ben Liblit. 2008. Reflections on the Role of Static Analysis in Cooperative Bug Isolation. In *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5079)*, María Alpuente and Germán Vidal (Eds.). Springer, 18–31. https://doi.org/10.1007/978-3-540-69166-2_2

[12] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: a manifesto. *Commun. ACM* 58, 2 (2015), 44–46. https://doi.org/10.1145/2644805

[13] MITRE. 2013. CVE-2013-1441. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1441 [accessed 09-March-2023].

[14] MITRE. 2018. CVE-2018-14876. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-14876 [accessed 09-March-2023].

[15] Raphaël Monat. 2021. *Static type and value analysis by abstract interpretation of Python programs with native C libraries. (Analyse statique, de type et de valeur, par interprétation abstraite, de programmes Python utilisant des librairies C).* Ph.D. Dissertation. Sorbonne University, Paris, France. https://tel.archives-ouvertes.fr/tel-03533030

[16] Alexander Patrakov. 2009. Dangers of setjmp()/longjmp(). https://patrakov.blogspot.com/2009/07/dangers-of-setjmplongjmp.html. [Online; accessed 09-March-2023].

[17] Eric S. Roberts. 1989. *Implementing Exceptions in C.* Technical Report 40. Digital Equipment Corporation Systems Research Center.

[18] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2019. PhASAR: An Inter-procedural Static Analysis Framework for C/C++. In *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11428)*, Tomás Vojnar and Lijun Zhang (Eds.). Springer, 393–410. https://doi.org/10.1007/978-3-030-17465-1_22

[19] Michael Schwarz, Simmo Saan, Helmut Seidl, Kalmer Apinis, Julian Erhard, and Vesal Vojdani. 2021. Improving Thread-Modular Abstract Interpretation. In *Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12913)*, Cezara Dragoi, Suvam Mukherjee, and Kedar S. Namjoshi (Eds.). Springer, 359–383. https://doi.org/10.1007/978-3-030-88806-0_18

[20] Michael Schwarz, Simmo Saan, Helmut Seidl, Julian Erhard, and Vesal Vojdani. 2023. Clustered Relational Thread-Modular Abstract Interpretation with Local Traces. In *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 13990)*, Thomas Wies (Ed.). Springer, 28–58. https://doi.org/10.1007/978-3-031-30044-8_2

[21] Helmut Seidl, Varmo Vene, and Markus Müller-Olm. 2003. Global invariants for analysing multi-threaded applications. In *Proceedings – Estonian Academy of Sciences Physics Mathematics*, Vol. 52. Estonian Academy Publishers, 413–436.

[22] Helmut Seidl and Ralf Vogler. 2022. Three improvements to the top-down solver. *Mathematical Structures in Computer Science* (2022), 1–45. https://doi.org/10.1017/S0960129521000499

[23] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. 2016. Static race detection for device drivers: the Goblint approach. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016.* ACM, 391–402. https://doi.org/10.1145/2970276.2970337

[24] Robert Paul Wilson. 1997. *Efficient, Context-Sensitive Pointer Analysis for C Programs.* Ph.D. Dissertation. Stanford University.

Received 2023-03-10; accepted 2023-04-21

HWASanIO: Detecting C/C++ Intra-object Overflows with Memory Shading

Konrad Hohentanner
konrad.hohentanner@aisec.fraunhofer.de
Fraunhofer AISEC
Garching, near Munich, Germany

Florian Kasten
florian.kasten@aisec.fraunhofer.de
Fraunhofer AISEC
Garching, near Munich, Germany

Lukas Auer
lukas.auer@aisec.fraunhofer.de
Fraunhofer AISEC
Garching, near Munich, Germany

Abstract

C/C++ are often used in high-performance areas with critical security demands, such as operating systems, browsers, and libraries. One major drawback from a security standpoint is their susceptibility to memory bugs, which are often hard to spot during development. A possible solution is the deployment of a memory safety framework such as the memory tagging framework Hardware-assisted AddressSanitizer (HWASan). The dynamic analysis tool instruments object allocations and inserts additional check logic to detect memory violations during runtime. A current limitation of memory tagging is its inability to detect intra-object memory violations i.e., over- and underflows between fields and members of `structs` and `classes`. This work addresses the issue by applying the concept of memory shading to memory tagging. We then present HWASanIO, a HWASan-based sanitizer implementing the memory shading concept to detect intra-object violations. Our evaluation shows that this increases the bug detection rate from 85.4% to 100% in the memory corruptions test cases of the Juliet Test Suite while maintaining high interoperability with existing C/C++ code.

CCS Concepts: • **Software and its engineering → Dynamic analysis**.

Keywords: memory safety, memory tagging, intra-object overflows, sub-object overflows, dynamic analysis

ACM Reference Format:
Konrad Hohentanner, Florian Kasten, and Lukas Auer. 2023. HWASanIO: Detecting C/C++ Intra-object Overflows with Memory Shading. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '23), June 17, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3589250.3596139

SOAP '23, June 17, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0170-2/23/06.
https://doi.org/10.1145/3589250.3596139

1 Introduction

The MITRE 2022 Top 25 Most Dangerous Software Weaknesses [5] lists memory errors in memory-unsafe languages, such as C and C++, among the leading causes for dangerous exploits. Vulnerabilities based on manual memory handling such as *out-of-bounds write/read* and *use after free* are ranked at positions 1, 5, and 7.

Sanitizers are dynamic analysis tools that can help to detect memory bugs during development by adding memory safety as part of compiler instrumentation. Hardware-assisted AddressSanitizer (HWASan) is a sanitizer available in the *gcc* and *clang* compilers, which uses memory tagging, also known as memory coloring, to improve bug detection. During object allocation, a tag is saved in the upper bits of pointers and tag memory associated with the object. On every access the tags are compared, with a mismatch leading to a runtime error.

One drawback of tagging-based sanitizers is their inability to detect intra-object violations i.e., over- and underflows between `struct` fields or `class` members, because the metadata layout handles objects as single entities without regarding the sub-objects they may contain.

In this work, we propose memory shading, a novel approach that splits the tag into color and shade. While the color part still indicates object boundaries, just like the tag in prior memory tagging approaches, the shade can now differentiate between fields and members of the same object. This allows the detection of intra-object violations and thus increases the bug-finding capabilities of tagging-based sanitizers, as we demonstrate in our prototype, HWASanIO.

In this paper, we make the following contributions:

- The memory shading concept, a modification of memory tagging, which expands the detection of temporal and spatial violations to also include intra-object violations.
- A full LLVM-based prototype implementation of HWASanIO that applies this concept to protect heap, stack, and global objects of C and C++ applications. [1]
- A detailed evaluation of HWASanIO and related solutions, specifically their effectiveness in detecting memory bugs and their performance overheads.

[1] Source code: https://github.com/Fraunhofer-AISEC/HWASanIO

2 Related Work

Compiler-based memory safety in C/C++ is possible through the use of metadata and runtime checks. This section gives an overview of available related work approaches and their respective security guarantees. Here, *spatial* memory safety ensures accesses are in-bounds, whereas *temporal* memory safety guarantees accesses are only permitted during an object's lifetime.

Interleaving *guard pages* [7, 20] or *red-zones* [4, 11, 12, 22, 24] between objects allows detection of linear overflows. Non-linear spatial violations can still occur for pointer offsets large enough to jump beyond a restricted area into another object. While these approaches can be relatively fast, they use a lot of additional runtime memory for the added restricted areas and the necessary shadowing of the application memory. They do not support intra-object detection, which would require insertion of metadata between object fields, and therefore break the Application Binary Interface (ABI).

Approaches with *bounds* metadata achieve memory safety by tracking the start and end address of objects. This can be done on a *per-object* basis, where pointer arithmetic is then instrumented to stay within the bounds of the corresponding allocations [6, 15, 21, 28], guaranteeing spatial memory safety. The precision of bounds-tracking solutions is increased with *per-pointer* metadata. *Fat-pointers* store this information directly inside the pointer [3, 14, 18, 25] to validate every spatial access. Related solutions such as Softbound/CETS keep the bounds information separated in memory and achieve temporal safety with *lock and key* metadata [16, 17, 19, 27]. Since pointers are compared to their start and end address, the check logic is quite complex. Tracing sub-object bounds is possible at the cost of an additionally increased overhead.

Some approaches deduce the object bounds from the pointer address by allocating objects in a bucketing scheme [1, 8, 10]. Because the metadata is solely defined by the object size it does not allow differentiation inside of aggregate types, and therefore offers no detection of intra-object violations. EffectiveSan [9] uses additional dynamic type information to detect spatial violations. This does include intra-object overflows, but not between fields with the same type, e.g., two `char` arrays inside a `struct`.

Memory tagging approaches [13, 23] such as HWASan use per-pointer tag metadata to achieve a low memory overhead and reduced overall check complexity compared to approaches with more complex metadata. At every access, the tag saved inside the pointer is compared for equality to the allocation tag stored in separate shadow memory directly linked to the object memory. Due to the limited tag size, metadata collisions are possible where a pointer to an object can access all other objects with the same allocation tag.

Listing 1. Intra-object overflow in Juliet CWE 121 stack-based buffer overflow testcase (variable names shortened).

```
#define STR "0123456789abcdef0123456789"
struct cV {
  char cFirst[16];
  void *vSecond;
  void *vThird; };
struct cV sCV;
memcpy(sCV.cFirst,STR,sizeof(sCV));
```

Dedicated hardware can be used to lower the runtime overheads of memory safety approaches. CHERI [26] uses so-called capabilities, i.e., fat pointers in hardware, to store bounds information with optional support for sub-bounds tracking, but requires considerable integration effort. ARM Memory Tagging Extension (MTE) [2] is a hardware-assisted memory tagging approach introduced as part of ARMv8.5. Tags are saved directly in hardware, with new instructions used for tag access. The load and store logic is able to automatically compare these tags, which greatly improves the performance compared to HWASan. The coarse granularity of tagging every 16 bytes of memory with only 4 bits of metadata leads to a higher collision rate between objects. Since the hardware extension only supports a single tag per object, no intra-object detection is possible.

In summary, current memory safety solutions only sparingly detect intra-object violations, with no compiler-based memory tagging solution being able to detect them.

3 Intra-object Memory Violations

An intra-object, or sub-object, memory violation occurs due to an over- or underflow between fields of the same object. An analysis of the source code used in the SPEC CPU 2017 benchmark, which includes compilers and image processing applications, shows that only 10% of all used aggregate variables are buffers, and 90% are `structs` and `classes`. With an average of 9 fields contained in these variables, the potential for intra-object violations is high. For current memory-tagging-based sanitizers such as HWASan, a detection is not possible because objects are seen as one unit during allocation with no differentiation between sub-objects.

Listing 1 shows a code snippet of a memory corruption test case from the Juliet test suite. Because the `sizeof` calculation is based on the whole `struct` and not the 16 byte buffer, the `memcpy` function is called with a too large size for its target. This results in an intra-object overflow and the corruption of all other data fields. By carefully overwriting neighboring variables, it is possible to either manipulate application data directly or, if the variables contain function or data pointers, start hijacking attacks.

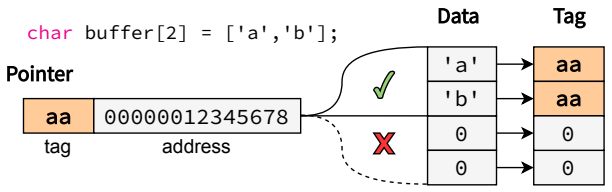


Figure 1. In memory tagging, tags are saved in pointer and object metadata to identify legal and illegal accesses.

4 Memory Tagging and HWASan

Memory tagging, or memory coloring, is a mechanism that can be used to enforce memory safety guarantees in an application by adding instructions to track a per-object tag in both the pointer address and dedicated metadata memory. Figure 1 shows this fundamental concept. During allocation, the buffer variable is assigned a random tag that is different from the previous and next memory object. The tag is added across the shadow memory range corresponding to the memory address of the buffer variable, as well as in the upper bits of the pointer. Every time this pointer is dereferenced, the two tags are compared to ensure accesses are in-bounds. The memory tagging implementation of HWASan included in the LLVM compiler infrastructure uses a flexible *granularity*, which refers to the number of bytes tagged with the same tag. In the default 16-to-1 mapping of HWASan, 16 bytes of application memory are associated with one byte of metadata. To be able to set a byte-precise end address inside this 16 byte range, a granule byte is added at the end of an object's metadata to indicate the last legally accessible byte. This requires special handling in the check logic. The metadata is saved in dedicated shadow memory mapped in the virtual address space; its location is derived from the application memory address. Pointers themselves include the tag in the 8 upper pointer bits, which are unused in virtual addressing and do not interfere with load and store instructions due to the ARM Top Byte Ignore (TBI) hardware feature.

From the application's point of view, the buffer variable is used normally during all operations, with the special case of load and store operations. Here, the compiler instrumentation adds instructions before each memory access to load the tag from the shadow memory address linked to the target address and compare it for equality with the tag stored inside the pointer. If the tags match, the operation continues normally. In the case of a mismatch, e.g., dereferencing a pointer incremented beyond the object's end address, an exception is thrown to indicate and prevent the memory violation. Additional debugging information, such as the stack frame and violation type, help to find the root cause of the memory corruption.

Memory tagging also allows the detection of temporal violations, because the shadow memory metadata of freed objects is set to a zero tag, invalidating all pointers to the

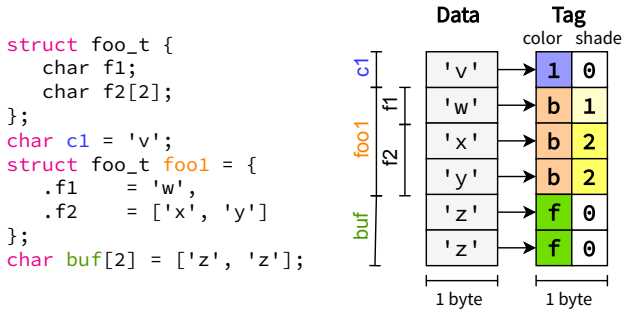


Figure 2. Memory shading splits the tag into color and shade to differentiate between objects and object fields.

object. Pointers to the now unallocated objects can therefore not be dereferenced anymore, preventing violations such as *use after free*.

5 Memory Shading

Memory shading expands the idea of memory tagging to offer protection against intra-object memory violations. To achieve this, the metadata design is changed from a purely color-based approach to a color- and shading-based approach, where the shade is used to differentiate between object members. Note: The shading concept covers both `structs` and `classes`, and both are protected in our prototype. To avoid redundancies we use `structs` in the following explanations and examples.

Color. The *color* is defined by the upper half of metadata. It is used in the same manner as the tag described in Section 4. During an object's allocation, it is set to a random non-zero value that does not collide with previous and following objects, which is ensured by comparing it with the color of neighboring metadata and generating new values until a differing value is found. The color is the same across the entire shadow memory range corresponding to the object and is used in all of the object's load and store checks.

Shade. The remaining bits of metadata are used as the *shade* of an object. It is different for neighboring fields, and thus allows detection of intra-object under- and overflows. For non-`struct` objects, it is set to zero, because any pointer to the object can legally access all its allocated memory. For `struct` objects, the shade starts at value one and is incremented for every additional non-`struct` field, wrapping around back to one after it reaches its maximum value. For layered `structs`, where a field is of a `struct` type, the shading algorithm is initiated anew, starting the shade of the field at value one. This keeps the shading layout consistent for objects with an arbitrary depth.

Example. Figure 2 shows an exemplary runtime memory layout of two variables on a system with the memory shading concept active. The C `struct` type `foo_t` contains a char

and a char buffer field. The metadata consists of a shared tag across the object and individual shades for each of the field variables. Additionally, a char and a buffer variable are allocated, which are not of a `struct` type and therefore have their shade set to zero, i.e., with no differentiation inside the object. Here, over- and underflows to other objects are prevented by the color stored in the metadata.

6 HWASanIO Implementation

The HWASanIO prototype is implemented as an extension to the HWASan source code available in the LLVM compiler framework (version 14), with support for Linux-based systems running on ARMv8 hardware. The following modifications are needed to add the support for memory shading to the HWASan instrumentation pass and its runtime library.

Metadata generation. Metadata in HWASanIO is generated with 4 bits for both color and shade, as discussed in Section 5. Here, the interceptors for heap management functions, as well as the stack and globals metadata handling have been adapted accordingly. Additionally, the metadata generation was modified to prevent tag collisions for neighboring objects by comparing against the previous and following tags in shadow memory. This ensures the detection of linear overflows and underflows.

Shadow Memory Layout. HWASan uses a 16-to-1 mapping, where every block of 16 bytes of data is shadowed by one byte of metadata. Object sizes therefore have to be increased to multiples of 16 bytes by adding padding bytes. This approach is not desirable for memory shading, because `structs` with additional padding between the fields would break the ABI and therefore dynamically linked uninstrumented library interfaces. Thus, HWASanIO uses a 1-to-1 mapping without padding. Shadow addresses are derived by flipping the Most Significant Bit (MSB) of the user-space virtual address.

Pointer instrumentation. An important aspect of color shading is the continuous update of the pointer shade. In HWASanIO, deriving a pointer to a sub-object sets the shade according to the targeted field. Here, the instrumentation pass inserts instructions for setting the corresponding shade whenever a sub-object pointer is created.

Check logic and optimizations. The check logic was modified to compare both color and shade stored in pointers and shadow memory, which leads to additional instructions being added. To reduce this overhead, new compile-time optimizations are introduced. A pointer flow analysis traces allocated objects that are never cast to `struct`-types and can therefore use the shorter, color-only, check on memory accesses. If an object's origin cannot be safely determined, the shade-aware check logic is used to avoid false negatives. This is the case for e.g., memory handling functions that take `void*` arguments and are used with both `struct` and non-`struct` arguments. For the SPEC CPU 2017 benchmark,

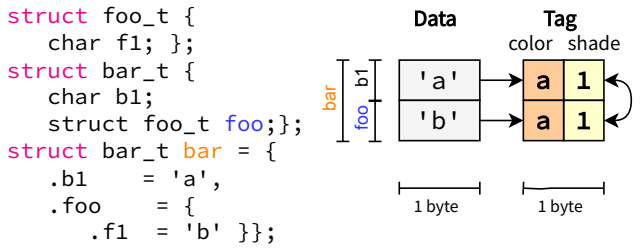


Figure 3. Shade collision in a layered `struct`.

this analysis safely simplified an average of 51% of checks to a color-only check.

Granule. HWASan uses a granule byte to mark the end address of an object on a byte-level granularity as discussed in Section 4. Because HWASanIO uses a 1-to-1 mapping and therefore always guarantees byte precision, the feature was removed without impacts to the memory safety guarantees.

Reporting. The reporting functionality of HWASan was enhanced to handle intra-object errors. The output contains the violation type and memory location of the corresponding variables as well as the usual debugging information such as stack trace and source code location.

Standard library wrappers. Compared to HWASan, the HWASanIO source code contains additional standard library wrappers for common memory handling functions such as *strcpy* to check the validity of the start and end of the source and destination arguments.

7 Security Discussion

In this section, we discuss the cases where HWASanIO is unable to detect memory violations during runtime, i.e., false negatives.

Due to the tag length, in some cases *tag collisions* are possible. While HWASanIO guarantees that neighboring allocations are assigned different colors, two non-neighboring objects may share a tag with a probability of approximately 7%. In this case, a non-linear access through a pointer to another object with the same tag is possible. This also applies to shade collisions of non-neighboring fields.

Collisions can also occur as a result of the algorithm assigning the shading values. To keep a consistent shade layout between all objects of the same `struct` type, field shades always start with value one and are then incremented. In the case of nested `structs`, this may lead to a shade collision for some layouts thereby preventing the detection of intra-object violations at either or both ends of a sub-object. An example is depicted in Figure 3. The `struct` variable `bar` contains a `char` variable and a `struct` field which consists of a single `char` variable. In this case, a neighboring over- or underflow between the fields, indicated by the arrow, cannot be detected, because the fields share the same shade.

Table 1. Juliet Test Suite vulnerability detection ratio.

CWE (# cases)	HWAIO (5364)	ASan (5364)	HWASan (5364)	Eff.San (5364)	Softb. (3970)
Stack Overflow	100%	96.7%	82.9%	55.8%	77.7%
Heap Overflow	100%	94.7%	94.6%	58.0%	73.7%
Buf. Underwr.	100%	100%	81.9%	100%	82.5%
Buf. Overrd.	100%	100%	99.7%	62.1%	96.5%
Buf. Underrd.	100%	100%	75.9%	98.0%	78.4%
Double Free	100%	100%	100%	0%	100%
Use After Free	100%	83.0%	50.9%	5.4%	51.3%
Free Inside Buf.	100%	100%	0%	0%	100%
Overall	100%	97.0%	85.4%	62.3%	78.9%

8 Evaluation

We evaluated both the bug detection capabilities and the performance and memory overheads of our HWASanIO prototype and approaches from related work. We compared our results with measurements of the unmodified HWASan [23] and the sanitizers ASan [22], EffectiveSan [9], and Softbound/CETS [16, 17]. They represent viable solutions for sanitizers using memory tagging, red-zones, dynamic typing, and bounds-checking, respectively. The tests were executed on an M2 MacBook Pro 2022 running Debian (Linux 5.19), except for EffectiveSan and Softbound/CETS which do not support ARM and were therefore evaluated on an AMD EPYC 7742 x86_64 system running Debian (Linux 5.10). Performance and memory overheads were measured with the SPEC CPU 2017[2] benchmark suite. For the functional evaluation, we used the Juliet Test Suite[3].

8.1 Vulnerability Detection Rate

The Juliet Test Suite is a collection of C/C++ test cases using Common Weakness Enumerations (CWEs) based on real-world applications. The test suite contains many temporal and spatial memory corruptions, which are always provided as a *good* version without any memory bugs and a *bad* version containing the memory corruption. Multiple variants of each test case are provided, where the same vulnerability exists in different control flow or data flow constructs, e.g., embedding the memory bug behind an *if*-condition.

For our evaluation, we removed variants which are difficult to automate because they depend on external or random input and only occasionally or never create an actual memory bug at run-time, resulting in a total of 5364 test cases. For the evaluation of Softbound/CETS, we excluded test cases that lead to compilation errors, resulting in a total of 3970 cases.

Table 1 lists the results of evaluating HWASanIO and related solutions with the Juliet Test Suite. None of the approaches showed crashes in the good variants of the test cases.

[2] https://www.spec.org/
[3] https://samate.nist.gov/SARD/test-suites/112

Importantly, HWASanIO is the only sanitizer with complete detection of all temporal and spatial bugs, and achieves a 100% vulnerability detection rate. AddressSanitizer (ASan) also achieves a very high detection rate of 97% and only misses violations in the intra-object overflow test cases and some *use after free* cases. Here, the Juliet Test Suite does not contain additional categories which would further differentiate the detection rate of HWASanIO from ASan such as non-linear overflows or additional variants of intra-object corruptions. Besides HWASanIO, only EffectiveSan can detect intra-object overflows between fields of differing types. Although Softbound/CETS has the conceptual support for it, the available implementation is incomplete in this regard. For HWASan, missing wrappers and an incomplete temporal bug detection account for most of the non-mitigated bugs. The dynamic typing used in EffectiveSan cannot detect many vulnerabilities in the overflow categories, because the test cases use buffers of the same type. While the bounds tracking used in Softbound/CETS should be able to detect most bugs, the released source code lacks features and is missing various wrapper functions.

In summary, HWASanIO is the only sanitizer able to detect all memory violations in the Juliet Test Suite. The Juliet Test Suite results therefore confirm the high bug detection capabilities of memory shading in practice.

8.2 Performance and Memory Overhead

To measure HWASanIO's overhead, we used the SPEC CPU 2017 benchmark suite, comparing the performance and memory consumption of benchmarks compiled with HWASanIO and other solutions to an uninstrumented baseline of the benchmarks. We compiled all benchmarks with optimization level 02 and executed them in single-threaded mode. Out of the 17 C/C++ *rate* benchmarks, we had to exclude *502.gcc* and *523.xalancbmk*, because of a memory violation detected by all sanitizers and *526.blender* due to undefined behavior detected by HWASanIO and EffectiveSan.

Softbound/CETS was ported from LLVM 3.9 to LLVM 9 to compile and run SPEC CPU 2017. Not all tests could be evaluated due to errors in the instrumentation.

Performance. The results of our performance evaluation are shown in Table 2. The evaluation shows that HWASanIO achieves an average performance overhead of 242.8%, which is around 90% slower than HWASan. When looking at the geometric mean, HWASanIO achieves a 129.1% overhead, which is close to the 118.9% overhead of HWASan. The additional overhead is largely due to the runtime of the `511.povray` benchmark, where the allocation of large numbers of local `struct` variables slows down the execution with HWASanIO. In the benchmark, the *Ray_In_Bound* function is executed an average of $8.12x10^{10}$ times per second and allocates a `struct` with 26 fields every time. For each of these fields a call to the shading function of HWASanIO is

Table 2. Measured performance overhead of memory safety frameworks for SPEC CPU 2017 benchmarks.

	Performance Overhead					Memory Overhead				
Benchmark	**HWASanIO**	**HWASan**	**ASan**	**Eff.San**	**Softb.**	**HWASanIO**	**HWASan**	**ASan**	**Eff.San**	**Softb.**
500.perlbench	367.8%	302.2%	90.4%	713.7%	-	146.8%	26.3%	429.1%	89.1%	-
505.mcf	81.8%	83.1%	38.6%	236.8%	584.9%	147.9%	6.6%	47.8%	3.5%	638.0%
508.namd	191.8%	260.9%	73.6%	131.5%	350.8%	110.5%	9.4%	204.9%	117.1%	88.3%
510.parest	185.3%	215.9%	66.8%	551.7%	-	154.5%	30.1%	458.5%	84.6%	-
511.povray	1607.6%	345.5%	153.6%	1284.7%	-	261.2%	115.0%	1018.8%	1499.8%	-
519.lbm	82.4%	37.0%	9.2%	131.0%	148.0%	100.1%	7.7%	13.8%	2.3%	1.9%
520.omnetpp	154.9%	88.4%	99.4%	203.6%	-	119.3%	27.4%	391.7%	71.6%	-
523.xalancbmk	127.3%	104.3%	69.4%	354.7%	-	148.2%	31.2%	563.7%	44.0%	-
531.deepsjeng	112.7%	127.2%	45.5%	90.9%	510.7%	99.9%	6.5%	1.3%	2.1%	2.2%
538.imagick	223.8%	204.9%	90.1%	195.8%	609.7%	133.2%	7.9%	186.2%	13.6%	106.5%
541.leela	133.0%	103.6%	62.1%	202.8%	-	100.2%	41.9%	7784.1%	387.1%	-
544.nab	84.5%	128.4%	43.9%	58.1%	243.4%	139.5%	29.0%	274.1%	70.1%	40.7%
557.xz	32.6%	49.4%	19.4%	78.0%	-	96.4%	4.4%	32.3%	3.8%	-
999.specrand	13.8%	39.5%	95.5%	12.4%	46.1%	154.0%	150.4%	444.5%	164.9%	390.9%
Average	242.8%	149.3%	68.4%	303.3%	356.2%	136.6%	35.3%	846.5%	182.4%	181.2%
Geometric Mean	129.1%	118.9%	56.6%	177.3%	270.5%	131.7%	19.9%	182.0%	38.6%	45.4%

required. A possible optimization to combat this issue is the preparation of the shading layout for the entire `struct` during instrumentation at the cost of an increased binary size instead of initializing every field individually. When looking at other benchmarks such as `557.xz`, HWASanIO runs faster than the original HWASan. This can be explained by the faster metadata access of the 1-to-1 metadata mapping as well as the simpler non-granule check logic for non-`struct` objects (see Section 6).

Memory Overhead. To compare the memory overhead we measured the peak memory consumption of the instrumented benchmarks with the GNU `time` command and compared the measurements with the non-instrumented baselines. Table 2 shows the average memory overheads of the evaluated approaches.

HWASanIO achieves an average memory overhead of 136.6%, compared to a 35.3% overhead for the original HWASan. This corresponds to the difference in mapping granularity used, where HWASanIO uses 1 byte of metadata per application byte and HWASan shadows 16 application bytes with a single tag byte. Since the 1-to-1 mapping is needed to conform with the ABI and the 8-bit metadata is already quite small, a reduction of the memory overhead is not possible.

Comparison. HWASanIO expands the bug detection capabilities at the cost of an additional memory and performance overhead. From a runtime performance perspective, the memory shading metadata management does not considerably impact the overall runtime behavior in most cases, except for applications that use large amounts of local `struct` variables. In comparison, ASan achieves lower runtime overheads than HWASanIO, which can be explained by its very efficient metadata setup and check logic, but it does not detect intra-object violations and has a larger memory overhead. The base version of HWASan has a significantly lower memory overhead but does not detect as many bugs, making it a good sanitizer choice for resource-constrained devices. EffectiveSan, the only other sanitizer which can detect intra-object violations between fields of different types, is slower than HWASanIO and requires more memory. Softbound/CETS is outperformed by HWASanIO in both memory and performance overhead. In summary, the new analysis capability is achievable with a manageable overhead increase.

9 Conclusion

We proposed the memory shading concept for C/C++ programs to allow intra-object detection for memory-tagging-based sanitizers. We implemented the HWASanIO prototype as an extension to HWASan, providing memory safety by protecting a program's heap, stack, and globals against both spatial and temporal memory corruptions and offering efficient intra-object detection. Since HWASanIO's design does not affect the underlying ABI it achieves a high interoperability level supporting existing C/C++ source code. We implemented a fully functional, LLVM-based HWASanIO prototype for ARM-based Linux systems which is available as compiler flag. In our security evaluation, the HWASanIO prototype detects more bugs than similar software-based solutions while maintaining a manageable performance and memory overhead for dynamic analysis. In summary, the HWASanIO prototype is a viable dynamic analysis tool to efficiently detect memory corruptions in C/C++ programs, including the previously hard-to-detect intra-object violations.

References

[1] 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 *Proceedings of the 18th USENIX Security Symposium (SEC '09)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity09/technical-sessions/presentation/baggy-bounds-checking-efficient-and

[2] ARM Limited. 2019. *ARM Architecture Reference Manual – ARMv8-A, for ARMv8-A architecture profile.* ARM DDI 0487E.a (ID070919).

[3] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. 1994. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, New York, NY, USA, 290–301. https://doi.org/10.1145/178243.178446

[4] Derek Bruening and Qin Zhao. 2011. Practical Memory Checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE, Washington, DC, USA, 213–223. http://dl.acm.org/citation.cfm?id=2190025.2190067

[5] MITRE Corporation. 2022. 2022 CWE Top 25 Most Dangerous Software Errors. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html. Accessed: 2023-03-15.

[6] Dinakar Dhurjati and Vikram Adve. 2006. Backwards-Compatible Array Bounds Checking for C with Very Low Overhead. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. ACM, New York, NY, USA, 162–171. https://doi.org/10.1145/1134285.1134309

[7] Dinakar Dhurjati and Vikram Adve. 2006. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In *Proceedings of the 2006 International Conference on Dependable Systems and Networks (DSN '06)*. IEEE. https://doi.org/10.1109/DSN.2006.31

[8] Gregory J. Duck and Roland H. C. Yap. 2016. Heap Bounds Protection with Low Fat Pointers. In *Proceedings of the 25th International Conference on Compiler Construction (CC '16)*. ACM, New York, NY, USA, 132–142. https://doi.org/10.1145/2892208.2892212

[9] Gregory J. Duck and Roland H. C. Yap. 2018. EffectiveSan: Type and Memory Error Detection Using Dynamically Typed C/C++. *SIGPLAN Not.* 53, 4 (jun 2018), 181–195. https://doi.org/10.1145/3296979.3192388

[10] Gregory J. Duck, Roland H. C. Yap, and Lorenzo Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS '17)*. Internet Society. https://doi.org/10.14722/ndss.2017.23287

[11] Niranjan Hasabnis, Ashish Misra, and R. Sekar. 2012. Light-Weight Bounds Checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. ACM, New York, NY, USA, 135–144. https://doi.org/10.1145/2259016.2259034

[12] Reed Hastings and Bob Joyce. 1991. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter 1992 USENIX Conference*.

[13] Konrad Hohentanner, Philipp Zieris, and Julian Horsch. 2023. CryptSan: Leveraging ARM Pointer Authentication for Memory Safety in C/C++. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing (SAC '23)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3555776.3577635

[14] 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*. USENIX Association, 275–288.

[15] Richard W M Jones and Paul H J Kelly. 1997. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In *Proceedings of the 3rd International Workshop on Automatic and Algorithmic Debugging (AADEBUG '97)*. Linköping University Electronic Press.

[16] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 245–258. https://doi.org/10.1145/1542476.1542504

[17] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM '10)*. ACM, New York, NY, USA, 31–40. https://doi.org/10.1145/1806651.1806657

[18] George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. ACM, New York, NY, USA, 128–139. https://doi.org/10.1145/503272.503286

[19] Harish Patil and Charles Fischer. 1997. Low-Cost, Concurrent Checking of Pointer and Array Accesses in C Programs. *Software: Practics and Experience* 27, 1 (Jan. 1997).

[20] Bruce Perens. 1999. Electric Fence Malloc Debugger. https://linux.die.net/man/3/libefence. Accessed: 2023-03-15.

[21] Olatunji Ruwase and Monica S. Lam. 2004. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS '04)*. Internet Society.

[22] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC '12)*. USENIX, Boston, MA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[23] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. 2018. Memory Tagging and How it Improves C/C++ Memory Safety. arXiv:cs.CR/1802.09517 https://arxiv.org/abs/1802.09517

[24] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *Proceedings of the 2005 USENIX Annual Technical Conference (ATC) (ATC '05)*. USENIX Association.

[25] Joseph L. Steffen. 1992. Adding Run-Time Checking to the Portable C Compiler. *Software: Practice and Experience* 22, 4 (1992).

[26] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy*. https://doi.org/10.1109/SP.2015.9

[27] Wei Xu, Daniel C. DuVarney, and R. Sekar. 2004. An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '04/FSE-12)*. ACM, New York, NY, USA, 117–126. https://doi.org/10.1145/1029894.1029913

[28] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. 2010. PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS '10)*. ACM, New York, NY, USA, 145–156. https://doi.org/10.1145/1755688.1755707

Received 2023-03-10; accepted 2023-04-21

Extensible and Scalable Architecture for Hybrid Analysis

Marc Miltenberger
marc.miltenberger@sit.fraunhofer.de
Fraunhofer SIT | ATHENE
Darmstadt, Hessen, Germany

Steven Arzt
steven.arzt@sit.fraunhofer.de
Fraunhofer SIT | ATHENE
Darmstadt, Hessen, Germany

Abstract

The prevalence of Android apps and their widespread use in daily life has made them a prime subject of study in program analysis. Apps for e-mail, navigation, mobile banking, eGovernment, healthcare, etc. each have their respective requirements on stability, efficiency, and security, which can be checked using static and dynamic analysis.

While developers and researchers can pick from a variety of scalable and integrated frameworks for static analysis, designing a dynamic analysis still requires significant engineering and design effort that contributes little to the analysis task at hand. Existing scholarly work on dynamic analysis has instead focused on individual challenges such as efficient data flow tracking, or code coverage in UI exploration. Combining dynamic analysis configuration and results with artifacts from static analysis is usually dealt with on an individual basis that does not generalize in the sense of a re-usable framework.

In this paper, we present a reference architecture and implementation for an integrated, scalable, and extensible hybrid analysis that offers a wide range of dynamic analysis capabilities. We hope that researchers can build upon our work for increased efficiency in hybrid analysis.

CCS Concepts: • **Software and its engineering → Dynamic analysis**.

Keywords: analysis, android, dynamic, static, architecture

ACM Reference Format:
Marc Miltenberger and Steven Arzt. 2023. Extensible and Scalable Architecture for Hybrid Analysis. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '23), June 17, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3589250.3596146

SOAP '23, June 17, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0170-2/23/06.
https://doi.org/10.1145/3589250.3596146

1 Introduction

With its market share of 72%, Android dominates the smartphone market. Users rely on their smartphones and the apps installed on them for a variety of tasks, many of which have domain-specific requirements on availability, efficiency, and security. Mobile banking apps process sensitive personal and financial data that must be protected against unauthorized access. Navigation apps must be efficient and available while the user is driving.

Android apps have therefore become a popular target for static and dynamic program analysis. For static analysis, developers and researchers can build novel approaches on top of existing frameworks such as Soot [12] with its Dexpler extension [5] or Wala [9]. Other libraries such as FlowDroid [4] or ICCTA [13] extend these framework with support for individual analyses, e.g., data flow, and the required Android-specific models. In summary, the frameworks provide readily-usable implementations of basic tasks, such as translating binary APK files into an intermediate representation (IR), computing a callgraph, or performing a data flow analysis. The user need not understand the details of, e.g., virtual dispatch or Android intent resolving as the framework abstracts from these technicalities and instead provides a higher level of abstraction. Users can, for example, query the data flow analysis or the callgraph, without needing to know how these data structures are generated by the framework.

For dynamic analyses, such a readily-usable framework does not exist. While Soot allows to modify the Jimple IR [19] and produce a new APK file from it, many building blocks are still missing. First, analysts must reason about low-level code changes for their task at hand. Secondly, dynamic analyses require devices, either virtual or physical. For scalability, device farms may be used [15]. Concurrently interacting with multiple devices, configuring emulators, installing and removing apps, keeping physical devices in a consistent state across multiple runs, etc. are recurring, yet non-trivial tasks for a dynamic analysis. Technical difficulties can lead to poor reliability due to, e.g., failing port forwarding. Lastly, the analysis must sign altered apps and, for network monitoring, must ensure that CA certificates for man-in-the-middle proxies are installed properly. We argue that these tasks should be abstracted through a framework, instead of being implemented individually for each analysis.

In addition to these setup tasks, communication between the app under analysis and the host computer must be maintained while the analysis is running. Though the contents of this communication are analysis-specific, the communication layer itself should be provided by the framework. As Android enforces strict requirements on the responsiveness of apps and terminates violators, special care must be taken to offload network handling from critical threads, while at the same time allow for timely data transmission. Frameworks should abstract from such runtime duties as well.

In this paper, we present AppRunner, an architecture and framework for dynamic analyses built on top of Soot. In addition to the basic infrastructure for the lifecycle management of instrumented apps and the corresponding devices as well as bidirectional communication between app and device, AppRunner provides building blocks for dynamic analyses. These building blocks include a dynamic value analysis, a dynamic callgraph, and a dynamic data flow analysis. Similar to their static counterparts in Soot, client code only needs to enable and configure the respective AppRunner modules. Client analyses can query the result data structures at any stage while the app is running or be notified when new values have been received from the device. Further, AppRunner maps all dynamic data back to Jimple units. Consequently, analyses only need to reason about a single abstraction of the code. It also enables, e.g., an integrated view on the callgraph that combines static and dynamic edges.

The remainder of this paper is structured as follows: In Section 2, we present the architecture of AppRunner. Section 3 provides an evaluation of AppRunner's basic performance. We present related work in Section 4 and conclude in Section 5.

2 Architecture

In this section, we describe the architecture and workflow of AppRunner.

2.1 General Workflow

AppRunner is built on Soot and relies on instrumentation to extract runtime data from apps. A hybrid analysis with AppRunner consists of five stages: initialization, static analysis, building, execution, and termination. In the following, we explain these stages in more detail.

In the *initialization stage*, the original APK file is read into Soot. In the following *static stage*, client analyses may conduct arbitrary static analyses using Soot. These steps are agnostic to the later dynamic analysis.

At the end of the static stage, the configuration for the dynamic stage is assembled. The configuration defines which dynamic analyses shall be instrumented into the app. By default, AppRunner supports a value analysis, a taint analysis, and a callgraph analysis. For the value analysis, the configuration also defines the logging points [18], i.e., the statements and the locals at these statements, for which values shall be obtained at runtime. Note that the configuration may depend on the outcome of a static analysis, e.g., by dynamically querying values that could not be computed statically.

In the *building stage*, AppRunner instruments the necessary changes into the app. Though client analyses may optionally define custom instrumentation steps, the core *instrumentation plan* is derived automatically from the configuration of the pre-defined dynamic analyses (value, callgraph, taint) supplied in the previous step. During building, AppRunner also instruments a pre-defined runtime framework into each app. The framework is responsible for maintaining the connection between the app and the analysis computer, and provides utility functions for the dynamic analysis parts instrumented into the app.

In the *execution stage*, this instrumented app is signed and installed on a device (physical device, emulator, or device farm) specified by the user. AppRunner establishes a connection with the app, receives runtime events, and passes them to the respective data structures (callgraph, taint graph, etc.). Client code may also register listeners to be notified immediately about new events. Supplying the app with input for exercising the user interface is an orthogonal problem, for which AppRunner integrates the Monkey tool from the Android SDK. Since AppRunner is agnostic to the UI driver, arbitrary other approaches can be integrated as well.

The final *termination stage* closes the connection between computer and app, and terminates and uninstalls the app from the device. In this stage, devices can be reset and post-processing of the collected data can be performed.

2.2 Detailed Architecture

Figure 1 shows the general architecture of AppRunner on the computer side. AppRunner strictly distinguishes between code that runs on the computer, code that runs on the device, and code that is shared between computer and device (mainly data objects). The device code may contain references to Android SDK classes that are unavailable in the JVM on the computer. Conversely, the computer implementation references Soot classes, which are not available on the phone.

On the computer, the *Runtime Value Notification Factory* (marked as ① in Figure 1) is the core component of each dynamic analysis. Note that AppRunner is fully extensible, i.e., a new notification factory can be added at any time. Each notification factory creates a *Runtime Value Notifier* ②. The notifier serves as a container for the logging points identified by the factory, i.e., the positions at which values shall be extracted. A logging point is a pair of statement and variable (unit and local in Soot's terminology). When the execution reaches the statement at runtime, the value of the variable is transmitted to the computer. If no local is given, an empty data item is transmitted, i.e., the logging point simply signals when the statement was reached.

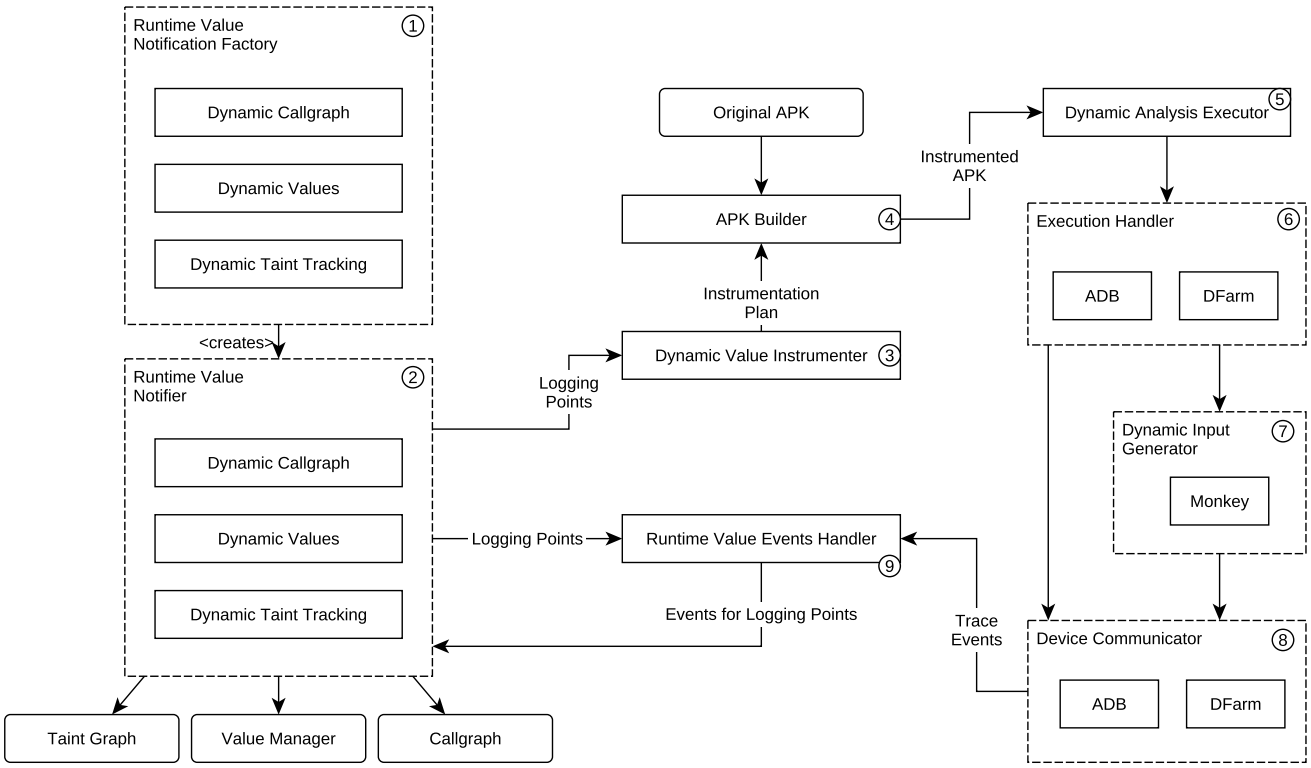


Figure 1. Architecture of AppRunner

The notifier provides the logging points to the *Dynamic Value Instrumenter* ③, which creates an *Instrumentation Plan*. The plan defines which statements must be injected at which positions to extract the values at the requested logging points. For later associating the values with the statements at which they were requested, the instrumentation plan associates each logging point with a unique ID. This ID also is instrumented into the app as a hard-coded parameter to the reporting call that is added for each logging point. When events are later received, they can be mapped by their ID. In addition to the logging point ID, each reporting call also takes the variable that shall be reported. Note that extracting values not only requires the runtime values to be serialized for transport. Extracting the values is a non-trivial task on its own, which we describe in Section 2.3.

The *APK Builder* ④ takes the original APK file, i.e., the current Soot scene, and the instrumentation plan to create an instrumented APK file, which is then passed to the *Dynamic Analysis Executor* ⑤. The executor identifies the *Execution Handler* ⑥ that supports the requested device. AppRunner supports one execution handler that covers physical devices and emulators, and one execution handler for a DFarm device wall. The execution handler implements a strategy to run the app. A strategy is not limited to a single run, but may also run the app multiple times to detect random behavior inside the app. Strategies may include interfaces to dynamic input generation tools such as Monkey ⑦ that explore the app's user interface to increase coverage. For interacting with the device, the execution handler and the input generator make use of the *Device Communicator* ⑧, which implements to low-level interactions with ADB (Android Device Bridge) or DFarm's REST API. AppRunner is extensible on the device side by adding more execution handlers and device communicators. The device communicator receives the *Trace Events* sent by the running app. On the app side, the instrumented code and the runtime framework are responsible for queuing trace events and sending them to the computer. Further, the communicator forwards the events to the *Runtime Value Events Handler* ⑨. The events handler is configured with a mapping between logging points and the runtime value notifier that has registered them. It uses this mapping to forward the events to the correct value notifier. It is upon the notifier to process the event according to the algorithm-specific semantics, e.g., extend the callgraph or process a taint.

2.3 Extracting Runtime Values

While the dynamic callgraph only needs to report logging point IDs, i.e., references to the statements that were reached, the value analysis also needs to report concrete values of arbitrary type. Since Android-specific classes (intents, bundles,

shared preferences, etc.) on the device cannot easily be serialized and transmitted to the computer, AppRunner's value analysis convert the values into a suitable representation on the device.

The instrumented reporting code invokes a method of the runtime framework, which inspects the runtime type of the object to transmit. It then queries a list of *Extraction Handlers*. Each handler checks whether it supports the respective type and, if so, translates the runtime object into an *Extraction Handler Data Object*. Depending on the type at hand, handlers may simply call getters to obtain the relevant data. In other cases, the data can only be accessed via reflection. Which data is semantically relevant is defined by the respective extraction handler. Transmitting all field values would unnecessarily increase data transmissions. Further, if the class is not available on the computer, it may be hard to process this low-level data without the corresponding implementation. The extractors therefore aim to collect (and, if necessary, compute) data in its final form.

Logging points may reference objects that have already lost all references to relevant data. A `FileInputStream`, for example, has no reference to the source file anymore once the constructor has returned. The numeric file descriptor from inside the app is not meaningful on the computer. For such special cases, AppRunner rewrites the app to wrap the target class with an *Interceptor Class* from the runtime framework. In the example of the `FileInputStream`, the interceptor class records the file name in its constructor and provides a getter for the extraction handler.

In total, AppRunner supports 42 different extraction handlers. To support more classes in the value analysis, AppRunner can easily be extended with new extraction handlers and data objects.

For handling dynamic code loading, special extraction handlers capture additional `dex` files that are loaded at runtime to extend the Soot scene on the computer. Further, a fallback extraction handler invokes `toString()` on each object referenced by a logging point. We found this fallback to be sufficient in most cases that are not modeled explicitly.

While normal logging points are fully defined at instrumentation time, a special extraction handler covers *Runtime Logging Points*. This handler attaches to all reflective method calls, evaluates the signature of the target method, and compares it to the registered runtime logging points. If a logging point definition matches the callee, the handler invokes the original extraction handler. The original handler receives the call arguments, which it can report as normal, i.e., without needing reflection support on its own.

2.4 Dynamic Taint Tracking

AppRunner's dynamic taint tracking is currently limited to objects, i.e., does not support tracking primitive values. Each object is referenced by its identity hash code. At each source, the respective hash code is transmitted to the computer. At each sink, AppRunner checks whether a tainted hash code is passed to the sink method. If so, a leak is reported.

Special handling is applied for container objects, i.e., objects that store tainted objects in one of their fields. If such a container object is passed to a sink, the above approach would miss a leak. AppRunner therefore instruments all field assignments to also taint the container object. The same handling applies to arrays and collections.

AppRunner's taint tracking requires all classes with assignments to be instrumented. The Android SDK and the Java standard library, however, are pre-loaded on the device and therefore not part of the app and cannot be instrumented. To model the effects of such library classes, AppRunner queries the StubDroid summaries [2] at instrumentation time if no code is available. At the call site inside the app code, it instruments code that mimics the taint transfers from the StubDroid summary.

2.5 Callgraph

AppRunner performs the computation of the dynamic callgraph on the host computer, since it may require a large amount of memory. Before method invocation sites as well as the start end exit points of methods, AppRunner places logging points with no special information except for the unique ID of the logging point and the thread ID of the thread triggering the logging point. When receiving the information that a logging point has been reached, using the instrumentation plan AppRunner knows which method the corresponding thread is. Since it remembers the previous call stack state of the thread, it can use that information to add an edge to the dynamic callgraph. To reduce the data consumption needed to transfer the callgraph information to the host computer, AppRunner groups a thousand events and sends them in bulk.

3 Evaluation

In this section, we evaluate AppRunner with respect to the following research questions:

RQ1 How scalable is AppRunner?
RQ2 How does AppRunner affect app reliability?
RQ3 Are the events of AppRunner correct?
RQ4 How does AppRunner affect the app size?

3.1 Experiment Setup

We evaluate AppRunner using a DFarm device wall equipped with 68 devices (58 Samsung XCover Pro, one Samsung S21, nine TP-Link Neffos C5L). Note that for each app DFarm chooses a device that is compatible with the respective app. The host system provides 64 logical Intel Xeon E5-4650 cores (released 2012) and 1 TB of DDR3 memory. We chose this system to run multiple experiments in parallel.

For our experiments, we ran AppRunner with 1,000 real-world apps obtained from the official Google PlayStore in

Table 1. Event throughput on the data set.

Type	Avg # of events per App	Avg bytes per event
Callgraph	5,261,461.44	25
Values	2,996.59	106.98
Taint	79,966.78	33.06
Avg per app	5,344,424.58	25.17

2022 via the AndroZoo repository [1]. For the taint analyses, we used the source/sink list available from the VUSC vulnerability scanner.

3.2 RQ1: Event Throughput

Each app from our dataset has been instrumented to run all supported analyses (value, callgraph, taint tracking). Each app has been exercised with monkey for five minutes.

Table 1 shows the average number of events and the average number of bytes transferred between device and computer per event. Apps on average trigger 5,344,424.58 events in total during the five minutes. This is equivalent to approximately 17814.75 events per second on average. Each app used 134.51 MB of event communication on average. We find that call graph events account for most of the traffic, but need the least space per event. We note that AppRunner was efficient enough to transfer and process all events, i.e., not a single event was dropped due to an event queue overflow. Note that AppRunner's runtime framework may theoretically drop events on the device in case they are queued too quickly for transmission to the computer and the queue overflows to avoid crashing the app.

3.3 RQ2: App Reliability

We checked that all apps instrumented by AppRunner could be installed on the devices, i.e., passed the Dalvik verifier. To check its function, we ran each app for five minutes and exercised it with the Monkey tool. No app crashed during the analysis. Further, no app was terminated by the operating system for failing the minimum responsiveness requirements despite the large number of events that was collected, sent, and processed.

3.4 RQ3: Correctness of Events

Too validate the correctness of the events obtained by AppRunner's integrated analyses, we randomly selected 100 edges from the dynamic callgraph across all apps. We manually verified these edges and found them to be correct. Note that the dynamic callgraph is more complex than the value analysis, because the edge is constructed from four individual events (call site, start of callee, end of callee, return site) and must cope with multi-threaded applications. AppRunner also supports special cases such as thread starting and Android executors.

3.5 RQ4: App Size Overhead

AppRunner instruments the app to integrate its dynamic analyses. The amount of Jimple statements added is different for the type of analyses. On average, for the callgraph 102,009 Jimple statements, for the taint analysis 174,055 and for the value analysis 840 statements are added per app. This is equivalent to an increase of 154 % per app. In addition, AppRunner injects its runtime components, which are loaded from class files during instrumentation time and which are the same for each target app. The runtime is equivalent to 6,110 statements. In total, each app grows by 157 % measured in Jimple statements. As the runtime is treated as an immutable component, AppRunner may inject runtime code that is not be needed by the current app at hand. We plan to remove such unused parts of the runtime code as future work. Nevertheless, we find that the extra statements do not negatively affect the apps.

4 Related Work

Previous work on dynamic analysis has mainly focused on individual tasks such as taint tracking [8, 21]. These works do not provide an extensible framework on which novel hybrid research work can be based and that allows for integrating user-defined static and dynamic client analyses.

With its pre-defined building blocks (callgraph, taint analysis, value analysis) and the concept of logging points, AppRunner provides a higher level of abstraction than function hooking framework such as FRIDA[1] or instrumentation frameworks such as Soot [3].

Other approaches use dynamic components to improve the precision or recall of static analyses [6, 7, 10]. These approaches provide a highly specific execution model and dynamic component, e.g., for taking heap snapshots, which is not generalizable for custom analyses. Further, JVM-based approaches lack support for the Android analysis lifecycle (sign, install, etc.).

AppRunner inherits some concepts such as logging points from earlier work on hybrid analysis [18]. At the same time, AppRunner relies on existing work for UI exploration and input generation as possible alternatives to Monkey [14, 16, 20]. AppRunner is intended for app analysis in a testing environment. Policy enforcement approaches [17], on the other hand, transform apps to check and enforce properties during normal use. These approaches feature a looser integration with host-based components (if any) and face higher performance requirements on the app.

Heid and Heider [11] structure a dynamic analysis framework into components: app stimulation, app behavior monitoring, control logic, and optionally network monitoring. They suggest manually combining tools for security analysis, which leaves substantial repeated effort with the analysis designer.

[1] https://frida.re/

5 Conclusion

In this paper, we have presented AppRunner, an extensible and scalable framework for conducting hybrid analyses on Android apps. AppRunner extends the Soot static analysis framework with a dynamic analysis engine. Out of the box, AppRunner supports dynamic callgraph, taint tracking, and value analysis. It can be extended with new analyses and more device types. In the future, we plan to extend AppRunner beyond Android to also support Java-based web applications running on popular application servers such as Tomcat or Glassfish. AppRunner provides the foundation of a dynamic analysis framework. In the future, we plan to investigate how dynamic analysis results affect the results of static analyses.

6 Data Availability

Our reference implementation is part of the VUSC [2] commercial vulnerability scanner. We are currently establishing an academic initiative to provide non-profit research institutions with free access to the source code of the implementation under a license that enables research while protecting our commercial interests in the scanner product. The evaluation and demo code can be found on Github [3].

Acknowledgments

This research work has been funded by the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

[1] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th international conference on mining software repositories*. 468–471.

[2] Steven Arzt and Eric Bodden. 2016. StubDroid: Automatic inference of precise data-flow summaries for the Android framework. In *Proceedings of the 38th International Conference on Software Engineering*. 725–735.

[3] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2017. The soot-based toolchain for analyzing android apps. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 13–24.

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[5] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. 2012. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*. 27–38.

[6] Manuel Benz, Erik Krogh Kristensen, Linghui Luo, Nataniel P. Borges, Eric Bodden, and Andreas Zeller. 2020. Heaps'n Leaks: How Heap Snapshots Improve Android Taint Analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1061–1072. https://doi.org/10.1145/3377811.3380438

[7] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*. 241–250.

[8] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 1–29.

[9] Stephen Fink and Julian Dolby. 2012. WALA–The TJ Watson Libraries for Analysis.

[10] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. 2017. Heaps don't lie: countering unsoundness with heap snapshots. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–27.

[11] Kris Heid and Jens Heider. 2021. Automated, Dynamic Android App Vulnerability and Privacy Leak Analysis: Design Considerations, Required Components and Available Tools. In *European Interdisciplinary Cybersecurity Conference*. 1–6.

[12] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, Vol. 15.

[13] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 280–291.

[14] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 224–234.

[15] Marc Miltenberger, Julien Gerding, Jens Guthmann, and Steven Arzt. 2020. Dfarm: massive-scaling dynamic android app analysis on real hardware. In *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*. 12–15.

[16] Nariman Mirzaei, Hamid Bagheri, Riyadh Mahmood, and Sam Malek. 2015. Sig-droid: Automated system input generation for android applications. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 461–471.

[17] Siegfried Rasthofer, Steven Arzt, Enrico Lovat, and Eric Bodden. 2014. Droidforce: Enforcing complex, data-centric, system-wide policies in android. In *2014 Ninth International Conference on Availability, Reliability and Security*. IEEE, 40–49.

[18] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting runtime values in android applications that feature anti-analysis techniques.. In *NDSS*.

[19] Raja Vallee-Rai and Laurie J Hendren. 1998. Jimple: Simplifying Java bytecode for analyses and transformations. (1998).

[20] Michelle Y Wong and David Lie. 2016. Intellidroid: a targeted input generator for the dynamic analysis of android malware.. In *NDSS*, Vol. 16. 21–24.

[21] Wei You, Bin Liang, Wenchang Shi, Peng Wang, and Xiangyu Zhang. 2020. TaintMan: An ART-Compatible Dynamic Taint Analysis Framework on Unmodified and Non-Rooted Android Devices. *IEEE Transactions on Dependable and Secure Computing* 17, 1 (2020), 209–222. https://doi.org/10.1109/TDSC.2017.2740169

Received 2023-03-10; accepted 2023-04-21

[2] https://www.sit.fraunhofer.de/en/offers/projekte/vusc/

[3] https://github.com/Fraunhofer-SIT/SOAP23-DynamicAnalysisArchitecture

User-Assisted Code Query Optimization

Ben Liblit
Amazon
USA

Yingjun Lyu
Amazon
USA

Rajdeep Mukherjee
Amazon
USA

Omer Tripp
Amazon
USA

Yanjun Wang
Amazon
USA

Abstract

Running static analysis rules in the wild, as part of a commercial service, demands special consideration of time limits and scalability given the large and diverse real-world workloads that the rules are evaluated on. Furthermore, these rules do not run in isolation, which exposes opportunities for reuse of partial evaluation results across rules. In our work on Amazon CodeGuru Reviewer, and its underlying rule-authoring toolkit known as the Guru Query Language (GQL), we have encountered performance and scalability challenges, and identified corresponding optimization opportunities such as, *caching*, *indexing*, and *customization of analysis scope*, which rule authors can take advantage of as built-in GQL constructs. Our experimental evaluation on a dataset of open-source GitHub repositories shows 3× speedup and perfect recall using indexing-based configurations, and 2× speedup and 51% increase on the number of findings for caching-based optimization.

CCS Concepts: • **General and reference** → **Performance**; *Experimentation*; • **Theory of computation** → **Automated reasoning**; **Programming logic**; • **Software and its engineering** → *Software defect analysis.*

Keywords: AWS, caching, GitHub, Guru Query Language (GQL), performance optimization, static analysis

ACM Reference Format:
Ben Liblit, Yingjun Lyu, Rajdeep Mukherjee, Omer Tripp, and Yanjun Wang. 2023. User-Assisted Code Query Optimization. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '23), June 17, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3589250.3596148

SOAP '23, June 17, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0170-2/23/06.
https://doi.org/10.1145/3589250.3596148

1 Problem Setting

Amazon CodeGuru Reviewer [10] is a commercial product that performs source-code repository scans as well as integrates into the code review process as an automated reviewer, leaving comments on pull requests. Its underlying architecture is based primarily on "micro-analyzers", which run narrow yet precise analysis scenarios. These are built atop a common abstraction layer, GQL, which contains reusable constructs such as forward/backward slicing, taint analysis, and filters to match code entities based on data types or call signatures.

GQL is our main vehicle to "democratize" CodeGuru Reviewer by empowering domain experts to directly specify, then tune and productionize, micro-analyzers. The GQL toolbox provides the building blocks for such micro-analyzers, which the expert then composes to express a property of interest, e.g. regarding correct usage of some cryptography or machine-learning library.

Our experience in supporting rule authors, and growing the CodeGuru Reviewer rule base, has exposed many cases where rules can — and in some cases, should — be optimized to run faster and make more frugal usage of compute and memory resources. This, in turn, has led us to design and implement several optimization features as part of the GQL toolbox, which are made available to rule authors to tune their rules' performance and resource consumption. In what follows, we set up the technical background for these optimizations, then describe them and report on their impact.

2 Background

CodeGuru supports Java and Python, and integrates with different code hosting platforms including GitHub and BitBucket. CodeGuru supports three code scanning modes:

- Incremental: A code review is created automatically when a pull request is raised.
- Full: The entire code base is analyzed upon request from a developer.
- CI/CD: The entire code base is analyzed as part of CI/CD workflows.

In any of the above modes, CodeGuru operates by (1) constructing an analysis-friendly *intermediate graph representation* of the target code base, then (2) applying a set of *rules* to

search for graph nodes in that representation that correspond to buggy code patterns.

2.1 Intermediate Graph Representation

CodeGuru's intermediate representation is the *MU graph* [21]. A MU graph is essentially a data-dependence graph overlaid with a control-flow graph, all in static single assignment (*SSA*) form. An individual MU graph node might represent a piece of data, an action that transforms input data into output data, or a control operation such as a branch. Nodes and edges carry additional details specific to their type and role. For example, a single action node that represents a function call might have:

- zero or more incoming data edges, each from some data node representing an argument to the call;
- an optional outgoing data edge, the target of which is some data node that receives the result of the call; and
- one incoming and one outgoing control edge, connected to the action or control nodes that execute immediately before or after this call.

The MU graph representation is language-independent. Actions are fairly fine-grained, and unnamed temporary values are made explicit. For example, the representation of print(a + b) would include a sum action node; a call action node; and three data nodes representing a, b, and the unnamed temporary value of a + b.

2.2 Rules

Finding buggy code patterns in a fine-grained MU graph can be cumbersome. To make this task easier, CodeGuru includes the Guru Query Language (*GQL*), a domain-specific language for operating on MU graphs [21]. A GQL rule consists of a sequence of operations on a set of MU graph nodes, called the *match frontier*. The match frontier is initially the set of all nodes in the MU graph representation of one function. GQL operations transform this set, such as by filtering it or by traversing the graph in a systematic way.

For example, one GQL operation might filter the match set to only the subset of data nodes that represent string literals. Another operation might transform each node in the match set to its data-flow successor. A higher-order operation could repeat the previous transformation while collecting a fixed-point. By chaining together these and a few more operations, one might create a rule that identifies all literal strings that can transitively flow into the second argument of a call to a function named "login". Thus, we have built a rudimentary rule that detects hard-coded passwords.

GQL is implemented as a Java library that relies heavily on the builder pattern. Starting with a fresh builder, one adds operations using calls like withDataByTypeFilter(...) or withOutNodesTransform(...). A final call to build() returns a constructed rule: an instance of GQL's CustomRule type that can be applied to functions or whole programs to

```
1  public void doPost(
2      HttpServletRequest servletRequest,
3      HttpServletResponse servletResponse,
4      FilterChain chain) {
5    String user =
6      servletRequest.getParameter("user");
7    String userPath = ".\\Data\\" + user;
8    ...
9    findUserDirectory(userPath);
10   ...}
11
12 private void findUserDirectory(String userPath) {
13   ...
14   File file = new File(userPath);
15   if (file.exists() && file.isDirectory()) {
16     String[] commands =
17         { "/bin/sh", "-c", "ls " + userPath };
18     Process process =
19         Runtime.getRuntime().exec(commands);
20     ...
21   }...}
```

Figure 1. Code snippet demonstrating Injection vulnerabilities

detect bugs. Unlike other rule-based static analysis languages such as, CodeQL [1], GQL does not require building the codebase (then compilation of the facts database), which limits adoption, blocks use cases like ad-hoc queries, etc. In terms of analysis capabilities, GQL offer codebase-wide data-flow and type-state capabilities, that is, deeper and more semantic analysis, unlike tool such as, Semgrep [2].

3 Motivating Example

To illustrate the insights feeding into the optimizations described in this paper, and the benefits that these optimizations introduce, we consider the code example in Figure 1, inspired by real-world code that our rules were evaluated on, where (untrusted) user input read via the getParameter call at line 6 reaches both the File constructor at line 14 and the exec call at line 19 through inter-procedural data flow. These flows give rise to path traversal and command injection vulnerabilities, respectively.

The corresponding CustomRule rule excerpt is shown in Figure 2. This rule detects taint-flow by using different optimization strategies, such as caching that caches various taint sources, configuration based indexing that dynamically indexes into a matching taint configuration, and specification of the tracking and analysis scope. In what follows is the discussion of the rule in Figure 2 using the code example in Figure 1.

```
1  CustomRule rule = new CustomRule.Builder()
2      ...
3      .withCachedDependency(b -> b
4        .withRuleConfigurationItemMatchFilter(
5          "$.Sources[*].method",
6          (n,c) -> n.isCall() && n.getName().matches(c))
7        .withInterproceduralDataDependentsTransform(
8            TrackingScope.FILE))
9      ...
10     .build();
```

Figure 2. Rule snippet demonstrating caching and configuration indexing

Caching. Injection vulnerabilities, such as those illustrated in Figure 1, are typically modeled as taint problems, where source/sink reachability is checked. The sources are often shared in common across multiple vulnerability categories, since these represent the reading of untrusted data into the program's state.

Caching provides a medium to exploit the following observation. Forward data-flow slices, starting from sources, can be computed once per function, then reused across other rules that agree on the sources as well as validators and sanitizers. In this case, reuse enables amortization across the path traversal and command injection rules.

The withCachedDependency statement at line 3 in Figure 2 illustrates this scenario. The subrule logic in the cached block reads sources from a configuration, then performs forward slicing from these sources.

Configuration indexing. Many rules are backed by a configuration, where the rule serves as a "template" that can be instantiated to model different code scenarios.

In a production setting, these configurations can reach the order of 10,000 entries, if not more, which mandates efficient handling. Brute-force iteration over the configuration to identify matching code entities (for example, source or sink calls) becomes prohibitive. We later describe an "inversion" of the configuration lookup, where an index is computed and code entities are then represented as keys enabling constant-time index lookup.

The withRuleConfigurationItemMatchFilter statement at line 4 in Figure 2 corresponds to this optimization. We omit the code to index into the configuration for space constraints, and instead focus on how the configuration is accessed. The first argument is a JSONPath query specifying which configuration items should be matched against entities in the code, whereas the second argument relates nodes n in the graph representation to configuration items c: in the example, call nodes whose name matches the configuration item.

Scope customization. In GQL, taint queries can be composed with other constructs, as well as instantiated in different ways at different points in the overall query. We have observed that in some cases, the return-on-investment from limiting the scope of a taint query, where we trade off time/resource costs versus recall, leans towards running the query on a smaller scope.

For the example in Figure 1, constraining the taint query to functions in the same file (while excluding functions in other files in the same repository) enables more precise analysis (less room for error, for example due to incorrect call resolutions), alongside faster and more resource-efficient analysis. The scope specification appears as the TrackingScope.FILE argument to the slicing operation at line 7 in Figure 2.

4 Optimization Strategies

4.1 Caching

The caching algorithm is based on a simple yet important observation. Given rules r_1 and r_2 with respective subrules sr_1 and sr_2, if

1. sr_1 and sr_2 are evaluated on equivalent states;
2. sr_1 and sr_2 perform the same operations; and
3. sr_1 and sr_2 both have sufficient analysis budget to complete their evaluation, or else both lack sufficient budget to complete their evaluation,

then the evaluation result due to sr_1 in the context of r_1 can be "reused" for sr_2 in the context of r_2, and vice versa. In what follows, We go over these criteria, and the meaning of "reuse", in turn.

Starting from the first criterion, a rule evaluation state consists of (i) the incoming match frontier, (ii) the match frontiers stored as variables (or IDs), and (iii) any additional metadata stored as part of the state. State equivalence reduces to equivalence along these three dimensions.

Rule as well as subrule isomorphism is checked in an inductive manner. Starting from the base case of atomic operations, these are compared directly. Composite operations, which consist of subrules and the operations therein (for example, withAnyOf or withAllOf), are compared starting from the subrules comprising them.

Finally, we check the analysis budgets attached to sr_1 and sr_2, where a budget is a bag of aspects, an aspect being a measurable "cost unit": wall-clock time, number of atomic analysis operations executed, number of functions visited during operation evaluation, and so on. We ensure that the budgets are *compatible*, in that both are simultaneously either sufficient to complete evaluation of sr_1 and sr_2, respectively, or both would be exhausted during subrule evaluation.

Assuming sr_1 and sr_2 are isomorphic and have compatible analysis budgets b_1 and b_2, respectively, the application of sr_1 to state σ_1 can be reused for sr_2 and σ_2 provided $\sigma_1 \equiv \sigma_2$, where by reuse, we mean that

1. the output state $\hat{\sigma}_1$ due to sr_1 is provided as the result of $[\![sr_2]\!]\sigma_2$; and
2. the budget cost recorded during evaluation of $[\![sr_2]\!]\sigma_2$ is deducted from sr_2's budget.

At the implementation level, the caching algorithm is built atop a thread-safe map. Map keys are rule/input pairs, where the values are the respective evaluation results. The caching algorithm checks, in an atomic block, whether the mapping is already established. If not, then the value is computed and inserted into the map.

While designed to be generic, caching is particularly useful when a potentially expensive subrule with a same set of matching frontier is embedded inside multiple rules. For example, taint tracking is a particularly helpful application of caching. Consider, as an example, distinct injection rules that share the same user input surface, thus same sources, yet differ in terms of sinks. The subrule that computes the forward slice from sources can be cached, hence amortized across all rules with only one of the rules performing the evaluation. This needs not be explicitly coordinated across the rules. Suffice it that they all wrap this evaluation step into a withCachedDependency statement, as shown at line 3 of Figure 2, and the reuse will emerge at run time. It is worth noting that caching is not free. There are performance overheads of writing to and reading from the cache. More importantly, there is a memory cost. Given that the memory used for caching is not unlimited, users shall use that memory to cache the expensive operations to optimize the performance gains of withCachedDependency.

4.2 Configuration Indexing

Analysis rules often cover multiple scenarios from one or more libraries. Examples include (i) flagging deprecated methods in the AWS Java API; (ii) tracking untrusted data from APIs that read user input; or (iii) checking that Closeable types are used correctly.

These are examples of rules backed by a configuration, listing the different instances that the rule logic applies to. In our experience, these configurations can reach the order of 10,000 entries, if not more. A naïve approach for evaluating configuration-backed rules is to iterate over all the configurations when evaluating the rule on a function f, for example by matching all calls made by f to a deprecated API, as listed in the configuration. For a configuration C, this means that the rule is evaluated, fully or in part, $|C|$ times on f. That is, evaluation time grows linearly with the size of the configuration.

We have designed and implemented an alternate scheme, where evaluation time is fixed irrespective of $|C|$. Our scheme stems from the observation that the configuration relates to entities in the code. Thus, we can start from the function under analysis, and relate entities therein to the configuration. As a simple example, for deprecated APIs, we can mine all the function calls in the function, and consult the configuration for any matches.

More generally, configuration indexing is backed by two functions provided by the rule author:

- An indexing function ι, mapping the configuration items $c \in C$ to key/value pairs $c \mapsto k$.
- A mapping function τ from entities in the code to the same domain of keys plus $\perp$ (for configuration-irrelevant entities).

Back to the example of deprecated APIs, the keys are the names of deprecated functions, i.e. ι projects deprecated API configurations — consisting of the AWS service, declaring class, and API name — on the API name as the key, whereas τ maps function calls within the target analysis scope to the callee name (and other code entities, like variables and control statements, to $\perp$). Thus ι, starting from configuration items, and τ, starting from the target scope, agree on how the configuration would be searched based on the code being analyzed: via function names.

With this "inversion", and assuming a good indexing function (such that there are few collisions, thus effective distribution across buckets), consulting the configuration requires nearly constant time regardless of its size. In practice, this has proven easy to achieve, since we typically make use of types and identifiers. The indexing and mapping functions are then both cheap to compute and yield effective distribution of configuration items.

5 Evaluation

In this section, we report on experimental evaluation.

5.1 Input Dataset

We have conducted the experiments on GitHub packages that have Apache or MIT licenses, and popularity of at least 4 stars. To evaluate the impact of different optimization strategies, we have selected two different datasets. The first dataset was used to evaluate configuration indexing optimization strategy. It consists of 200 randomly selected Java and Python GitHub repositories which have specific SDK usages, such as AWS Java SDK [9] or AWS Python SDK [8]. The second dataset was used to evaluate different caching strategies and analysis scopes. It consists of another 180 randomly selected Java GitHub repositories which have specific APIs that are identified as tainted sources. The average number of lines of code in repositories from the dataset is 25697.

5.2 Experimental Setup

The experiments were run on an Amazon EC2 machine with 48 cores, 384 GB of memory, and 2 hard drives of size 1 TB each. We have selected 5 AWS best practice rules and 7 taint-flow rules to demonstrate the impact of the configuration indexing, and caching, respectively. Depending on the usage scenarios, users could have different requirements about

time limits to run the analysis. For example, an offline scan could have a longer time limit, while an online scanning during Code Review typically demands a shorter time limit. We evaluated our rules on open-source GitHub packages, with a time limit of 30 minutes and 5 minutes per package.

5.3 Experiment 1: Configuration Indexing

Table 1 presents the impact of indexing based configuration using 5 CodeGuru rules [24], that specifies a set of guidelines for correct, secure, and performant usage of AWS cloud Java and Python SDKs.

Column 1 in Table 1 gives the rule id, and Column 2 presents the total number of configurations that each rule evaluates on. Columns 3–4 report the run times and number of findings or detection from the rules *without* indexing optimization. Columns 5–6 report the same *with* indexing optimization. Comparing the run times of the rules without indexing and with indexing in Table 1, it is evident that when the total number of configurations are large (>1,000), the unoptimized rules without indexing, *Rule 1* and *Rule 2*, timed out. The evaluation time of the unoptimized rules grow linearly with the size of the configuration that the rules operate on. For rules that evaluate on few hundred configurations, such as *Rule 3*, *Rule 4*, and *Rule 5*, the speedup is 3× or more. Furthermore, the number of findings (reported in #Findings) show that the unoptimized rules, *Rule 1* and *Rule 2*, did not produce any findings, while the optimized rules produced same number of findings for different time limits. This demonstrates that the dynamic indexing of configurations help uncover more number of bugs overall.

5.4 Experiment 2: Caching and Scope Customization

In this experiment, we evaluated the impact of caching and scope customization. We ran seven rules, targeting different kinds of injection vulnerabilities, including command injection, SQL injection, cross-site scripting, log injection, path traversal, LDAP injection, and XPath injection [24]. All the rules shared a same set of tainted sources, the vast majority of which represent data coming from the Internet and is generally considered to be untrusted input to the program. Depending on the injection issue, the rules differ in sinks. For example, the rule for command injection considers APIs responsible for OS command execution as sinks. We used various combinations of analysis scopes (i.e., file-level and package-level) and caching configurations (i.e., with and without caching) on these rules. We ran these rules against 180 repositories in the second dataset with a time limit of 5 minutes and 30 minutes.

The results based on a time limit of 5 minutes are shown in Table 2. In this table, column 1 indicates whether the static analyzer performed a whole-program inter-procedural analysis, i.e, package-level, versus, a more contained file-level inter-procedural analysis. Columns 2–3 list the caching configuration we set for each rule. We experimented with different cache sizes, which specify the maximum number of cached tainted program points. Column 4 presents the number of cache hits and misses for each experiment. Under columns 5–9, we first list the evaluation time for all the rules, and then for the first rule, and then for the rest of the rules. The reason of splitting the rules in this way is to show the effect of caching. We also present the average and median evaluation time it takes for analyzing a repository. Column 10 summarizes the number of findings we obtained for all the rules. Column 11 reports the number of repositories that our analysis timed out on the given time limit. Due to the space constraint, we did not list the results based on a time limit of 30 minutes. We will discuss about the numbers during comparison.

Impact of scope customization: Our evaluation results demonstrate the importance of customizing the analysis scope. When caching is unavailable, file-level analysis scaled well on a time limit of 5 minutes. Comparing to package-level analysis, the speedup of the total rule evaluation time was more than 1.5×, which also reduced the number of timeouts from 17 repositories to 1. As for the number of findings, the file-level analysis only reported 29 fewer findings (8% less) than the package-level analysis. These numbers suggest that given a short time limit, even if we enabled package-level analysis, the analysis was not able to scale properly without caching. On the other hand, file-level analysis performed well in terms of meeting the time limit without sacrificing too much recall.

If users have more budgets in terms of time limit and are willing to increase the limit to 30 minutes, we observed obvious improvement on recall using package-level inter-procedural analysis. Even without caching, comparing to file-level analysis, the number of findings increased 56% from 339 to 529. When caching is in place, the improvement is even more significant, as we discuss below.

Impact of caching: Results show that caching can significantly improve the recall of package-level inter-procedural analysis. When the time limit is 5 minutes, the speedup of overall rule execution was more than 1.7×, comparing to no caching. This directly resulted in an increased number of findings by 44% from 343 to 495. When the time limit is 30 minutes, the speedup was even increase to 2×. The number of findings was increased by 51% from 529 to 799. Caching is effective even when the time limit is 5 minutes. Comparing to no caching, only the analysis time of the first rule slightly increased, likely caused by the overhead from cache writes. Such overhead was well offset by later-on savings when the rest of the rules were executed. If users have more budgets on memory, they can increase the cache size and maximize the benefits of caching. Looking at the last row of Table 2 where a cache size of 100,000 was used, the six rules that can make use of cached entries in total only took 15% more analysis time than the time of the first rule alone. The overall

Table 1. Comparison for rules with and without configuration indexing. Evaluation times are in seconds, given as "x / y" for for 30-minute and 5-minute limits, respectively.

Rule	# Configurations	Without Indexing		With Indexing	
		Evaluation Times	# Findings	Evaluation Times	# Findings
		30 mins / 5 mins		30 mins / 5 mins	
Rule 1 [6]	1,117	Timeout / Timeout	N/A	215.3s / 215.3s	78
Rule 2 [3]	8,411	Timeout / Timeout	N/A	296.7s / 296.7s	136
Rule 3 [5]	81	427.7s / Timeout	32	144.5s / 144.5s	32
Rule 4 [4]	186	694.5s / Timeout	56	139.4s / 139.4s	56
Rule 5 [7]	126	673.1s / Timeout	47	126.4s / 126.4s	47

Table 2. Comparison for rules with and without caching at file or package scope with time limit of 5 minutes.

	Cache			Analysis Time (in seconds)						
Scope	Configuration	Size	# Hits/Misses	All Rules	First Rule	The Rest	Mean	Median	# Findings	# Timeouts
File	Disabled	N/A	N/A	2,796.4	443.0	2,353.4	15.7	2.7	314	1
File	Enabled	10,000	39,800/6,653	2,730.9	442.0	2,288.9	15.3	2.6	314	1
File	Enabled	100,000	39,800/6,653	2,725.8	440.3	2,285.5	15.3	2.7	314	1
Package	Disabled	N/A	N/A	4,327.6	1,121.3	3,206.3	39.1	3.7	343	17
Package	Enabled	10,000	20,565/5,597	2,679.9	1,138.5	1,541.4	28.4	3.0	406	12
Package	Enabled	100,000	21,705/4,651	2,448.8	1,134.5	1,314.3	26.9	2.9	495	11

speedup helped to discovered 44% more findings given the short time limit.

Caching was even more effective when the time limit is 30 minutes, comparing to the time limit of 5 minutes. The speedup and the percentage of increased number of findings both improved. When the time limit was longer, the first rule had its chance to finish on more complex repositories and wrote to the cache. Due to complex taint flows in these repositories, a larger cache size was needed otherwise the cache can only hold a portion of the tainted program points. Once the required resources on both time and memory were met, users can maximize the benefits of caching.

6 Related Work

Toman and Grossman [26] note caching as widely used technique to make static analysis tractable [11, 12, 18–20, 22, 25], but limited to reanalysis of the same program or of shared library code [17]. Prior work on analysis caching has generally keyed the cache on coarse-grained program components, such as functions or files. By contrast, we can cache results of whole rules, subrules, or even individual GQL operations. Our approach is well-matched to a feature-rich analysis service that checks many aspects of a single code base [24], as our cache can accelerate common intermediate steps across multiple rules. Our focus on efficiently applying many checks to varied programs contrasts with, and is complementary to, that of Gu et al. [13], who focus on scaling any single analysis to large programs.

Toman and Grossman [26] propose a community database of analysis-relevant API information. If this effort succeeds, then the sheer number of annotated APIs may become a scaling challenge. We have shown that configuration indexing works well for rules that operate with thousands of configurations, that are mined from different SDKs.

Schubert et al. [23] discuss the importance of understanding analysis performance so that it can be tuned to perform well. Toman and Grossman [26] also note the use of tunable "knobs" to balance precision and performance [14–16]. Our analysis scopes are one such group of knobs, but we have not detailed a procedure for selecting the best scopes for any given task. The instrumentation-directed strategies of Schubert et al. [23] are likely applicable here.

7 Conclusion

In this paper, we have presented an interactive approach for rule authors—encoding their domain expertise as GQL rules evaluated through Amazon CodeGuru Reviewer—to optimize their rules' performance. Specifically, rule authors can (i) cache rule steps for reuse by co-evaluated rules; (ii) control the scope of interprocedural queries at a granular level; as well as (iii) scale a rule "template" to a large number of configurations using efficient indexing. Our evaluation of these optimizations on a GitHub dataset indicates significant performance gains, e.g. ×3 speedup thanks to configuration indexing and ×2 speedup thanks to caching.

References

[1] 2019. CodeQL. https://codeql.github.com

[2] 2020. Semgrep. https://semgrep.dev

[3] 2022. CodeGuru Rule: Batch request with unchecked failures. https://docs.aws.amazon.com/codeguru/detector-library/java/aws-unchecked-batch-failures/.

[4] 2022. CodeGuru Rule: Check uncaught exceptions High. https://docs.aws.amazon.com/codeguru/detector-library/java/check-uncaught-exceptions/.

[5] 2022. CodeGuru Rule: Inefficient polling of AWS resource High. https://docs.aws.amazon.com/codeguru/detector-library/java/aws-polling-instead-of-waiter/.

[6] 2022. CodeGuru Rule: Missing pagination. https://docs.aws.amazon.com/codeguru/detector-library/java/missing-pagination/.

[7] 2022. CodeGuru Rule: Use of a deprecated method. https://docs.aws.amazon.com/codeguru/detector-library/java/deprecated-method/.

[8] Amazon Web Services. [n. d.]. AWS SDK for Python (Boto3). https://aws.amazon.com/sdk-for-python/

[9] Amazon Web Services. [n. d.]. Boto3 - The AWS SDK for Java. https://github.com/aws/aws-sdk-java

[10] Amazon Web Services. [n. d.]. What is Amazon CodeGuru Reviewer? https://docs.aws.amazon.com/codeguru/latest/reviewer-ug/welcome.html

[11] Steven Arzt and Eric Bodden. 2014. Reviser: efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 288–298. https://doi.org/10.1145/2568225.2568243

[12] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 459–465. https://doi.org/10.1007/978-3-642-20398-5_33

[13] Rong Gu, Zhiqiang Zuo, Xi Jiang, Han Yin, Zhaokang Wang, Linzhang Wang, Xuandong Li, and Yihua Huang. 2021. Towards Efficient Large-Scale Interprocedural Program Static Analysis on Distributed Data-Parallel Computation. *IEEE Trans. Parallel Distributed Syst.* 32, 4 (2021), 867–883. https://doi.org/10.1109/TPDS.2020.3036190

[14] Ben Hardekopf, Ben Wiedermann, Berkeley R. Churchill, and Vineeth Kashyap. 2014. Widening for Control-Flow. In *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8318)*, Kenneth L. McMillan and Xavier Rival (Eds.). Springer, 472–491. https://doi.org/10.1007/978-3-642-54013-4_26

[15] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: a static analysis platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 121–132. https://doi.org/10.1145/2635868.2635904

[16] Yoonseok Ko, Hongki Lee, Julian Dolby, and Sukyoung Ryu. 2015. Practically Tunable Static Analysis Framework for Large-Scale JavaScript Applications (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 541–551. https://doi.org/10.1109/ASE.2015.28

[17] Sulekha Kulkarni, Ravi Mangal, Xin Zhang, and Mayur Naik. 2016. Accelerating program analyses by cross-program training. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 359–377. https://doi.org/10.1145/2983990.2984023

[18] Yingjun Lyu, Sasha Volokh, William G. J. Halfond, and Omer Tripp. 2021. SAND: a static analysis approach for detecting SQL antipatterns. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 270–282. https://doi.org/10.1145/3460319.3464818

[19] Scott McPeak, Charles-Henri Gros, and Murali Krishna Ramanathan. 2013. Scalable and incremental software bug detection. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, Bertrand Meyer, Luciano Baresi, and Mira Mezini (Eds.). ACM, 554–564. https://doi.org/10.1145/2491411.2501854

[20] Rashmi Mudduluru and Murali Krishna Ramanathan. 2014. Efficient Incremental Static Analysis Using Path Abstraction. In *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8411)*, Stefania Gnesi and Arend Rensink (Eds.). Springer, 125–139. https://doi.org/10.1007/978-3-642-54804-8_9

[21] Rajdeep Mukherjee, Omer Tripp, Ben Liblit, and Michael Wilson. 2022. Static Analysis for AWS Best Practices in Python Code. In *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany (LIPIcs, Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 14:1–14:28. https://doi.org/10.4230/LIPIcs.ECOOP.2022.14

[22] Lori L. Pollock and Mary Lou Soffa. 1989. An Incremental Version of Iterative Data Flow Analysis. *IEEE Trans. Software Eng.* 15, 12 (1989), 1537–1549. https://doi.org/10.1109/32.58766

[23] Philipp Dominik Schubert, Richard Leer, Ben Hermann, and Eric Bodden. 2019. Know your analysis: how instrumentation aids understanding static analysis. In *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, Neville Grech and Thierry Lavoie (Eds.). ACM, 8–13. https://doi.org/10.1145/3315568.3329965

[24] Amazon Web Services. 2023. CodeGuru Rules. https://docs.aws.amazon.com/codeguru/detector-library/.

[25] Amie L. Souter and Lori L. Pollock. 2001. Incremental Call Graph Reanalysis for Object-Oriented Software Maintenance. In *2001 International Conference on Software Maintenance, ICSM 2001, Florence, Italy, November 6-10, 2001*. IEEE Computer Society, 682–691. https://doi.org/10.1109/ICSM.2001.972787

[26] John Toman and Dan Grossman. 2017. Taming the Static Analysis Beast. In *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA (LIPIcs, Vol. 71)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 18:1–18:14. https://doi.org/10.4230/LIPIcs.SNAPL.2017.18

Received 2023-03-10; accepted 2023-04-21

Completeness Thresholds for Memory Safety of Array Traversing Programs

Tobias Reinhard
tobias.reinhard@kuleuven.be
imec-DistriNet, KU Leuven
Leuven, Belgium

Justus Fasse
justus.fasse@kuleuven.be
imec-DistriNet, KU Leuven
Leuven, Belgium

Bart Jacobs
bart.jacobs@kuleuven.be
imec-DistriNet, KU Leuven
Leuven, Belgium

Abstract

We report on intermediate results of – to the best of our knowledge – the first study of *completeness thresholds* for (partially) bounded memory safety proofs. Specifically, we consider heap-manipulating programs that iterate over arrays without allocating or freeing memory. In this setting, we present the first notion of completeness thresholds for program verification which reduce *unbounded* memory safety proofs to (partially) *bounded* ones. Moreover, we demonstrate that we can characterise completeness thresholds for simple classes of array traversing programs. Finally, we suggest avenues of research to scale this technique theoretically, i.e., to larger classes of programs (heap manipulation, tree-like data structures), and practically by highlighting automation opportunities.

CCS Concepts: • **Theory of computation → Program verification**; **Separation logic**; **Verification by model checking**.

Keywords: program verification, completeness thresholds, memory safety, bounded proofs, model checking, separation logic

ACM Reference Format:
Tobias Reinhard, Justus Fasse, and Bart Jacobs. 2023. Completeness Thresholds for Memory Safety of Array Traversing Programs. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '23), June 17, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3589250.3596143

1 Introduction

Unbounded vs Bounded Proofs. Many techniques have been developed to convince ourselves of the trustworthiness of software. A fundamental pillar for any higher-level property is memory safety. In memory-unsafe languages the burden of proof lies with the programmer. Yet, it remains hard to prove and in general requires us to write tedious, inductive proofs. One way to automate the verification process is to settle on bounded proofs and accept bounded guarantees.

Consider a program c that searches through an array of size s. An unbounded memory safety proof for c would yield that the program is safe for any possible input, in particular for any array size, i.e., $\forall s.\ \texttt{memsafe}(c(s))$. A bounded proof that only considers input sizes up to ten would only guarantee that the program is safe for any such bounded array, i.e., $\forall s \leq 10.\ \texttt{memsafe}(c(s))$.

Completeness Thresholds. Approximating unbounded proofs by bounded ones is a technique often used in model checking. Hence, the relationship between bounded and unbounded proofs about finite state transition systems has been studied extensively [1, 4, 7, 13, 20, 30, 35, 38]. For a finite transition system T and a property of interest ϕ, a *completeness threshold* is any number k such that we can prove ϕ by only examining path prefixes of length k in T, i.e., $T \models_k \phi \Rightarrow T \models \phi$ [20] [1]. Over the years, various works characterised over-approximations of least completeness thresholds for different types of properties ϕ. These over-approximations are typically described in terms of key attributes of the transition system T, such as the *recurrence diameter* (longest loop-free path) [35]. For instance, consider the class of global safety properties of the form Gp for finite transition systems T, where p is a local property. We know that the smallest completeness threshold for this class expressible solely in terms of T's diameter is exactly $\texttt{diam}(T)$ [7, 34]. As safety property of the form Gp, this also applies to memory safety of finite transition systems.

In general, heap-manipulating programs' state space can be infinite. That is because the program's input data can be arbitrarily large and because executions can be arbitrarily long. Therefore, the key attributes described above will generally be infinite as well. This vast structural difference between the programs we are interested in and the transition systems for

SOAP '23, June 17, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0170-2/23/06…$15.00
https://doi.org/10.1145/3589250.3596143

[1] Note that the term completeness threshold is used inconsistently in literature. Some papers such as [20] use the definition above, according to which completeness thresholds are not unique. Others like [35] define them as the minimal number k such that $T \models_k \phi \Rightarrow T \models \phi$, which makes them unique.

```
for i in [L : s-R] do
  !a[i+Z]
```
$\Big\}$ =: $trav^Z_{L,R}(a,s)$

$Trav \quad := \quad \{trav^Z_{L,R} \quad | \quad L,R,Z \in \mathbb{Z}\}$

Figure 1. Class *Trav* of programs $trav^Z_{L,R}$ traversing an array a of size s, attempting to read elements. L, R, Z are constants.

which completeness thresholds have been studied prevents us from reusing any of the existing definitions or results.

2 Limitations of Bounded Proofs

Bounded Model Checking. Generally, if we want unbounded memory safety guarantees, we have to consider all possible input sizes and all possible executions. This is often hard and requires us to write tedious inductive proofs. An alternative is to give up on the idea of unbounded guarantees and to settle for bounded ones. One approach that has proven useful during development of critical software is bounded model checking (BMC) [15].

The underlying idea is to approximate the original verification problem by a finite model that we can check automatically. With this approach, we choose a size bound S and only consider inputs with sizes up to S. Further, we also only check finite execution prefixes. A common approach is to unwind loops and recursion up to a certain depth.

The intuition behind this approach is that if the program contains errors, they likely already occur for small input sizes and early loop iterations. As long as BMC does not perform abstraction [16], all reported counterexamples are real bugs. However, we should be careful not to forget that this way we only obtain a bounded proof yielding bounded guarantees.

Array Traversal Pattern. Consider the class of programs *Trav* presented in Fig. 1 in a WHILE language with pointer arithmetic. Given a pointer a and a variable s, such that a points to an array of size s, each program iterates through the array and attempts to read elements. The class models a basic programming pattern and common off-by-n errors [23]. We use upper case letters for constants and lower case letters for (program) variables. A program $trav^Z_{L,R}$ from this class iterates from $i = L$ to $i = s - R$ (bounds incl.) and attempts to read the array at index $i+Z$. We use $!x$ to express accesses to a heap location x. Whether memory errors occur for a concrete instance $trav^Z_{L,R}$ depends on how the constants L, R, Z are chosen. We use it as minimal example throughout this paper.

What Could Go Wrong with Bounded Proofs? To illustrate the issue, let us use BMC to check various instances of the array traversal pattern: (i) traversal of the entire array: $trav^0_{0,1}$, (ii) traversal of the array with accesses offset by two from the index: $trav^2_{0,1}$ and (iii) an additional reduction of the index variable's upper bound by one: $trav^2_{0,2}$. It is easy to see that (i) is memory-safe while (ii) and (iii) are not. However, before we run a model checking algorithm we have to choose appropriate bounds. The pattern we are looking at is quite simple. So, we choose size bound $S = 1$ and unwinding depth $D = 1$ for the BMC procedure. Note that the latter effectively means: we do not restrict the loop depth for the input sizes we chose.

For the standard variant (i) we cannot find any errors within the bounds. This is fine because the program is safe. In variant (ii) array accesses $a[i + 2]$ are incorrectly shifted to the right. This already leads to an out-of-bounds error for arrays of size 1. This size falls within our chosen bounds, so BMC reports this error and we can correct it. Finally, (iii)'s reduction of the index variable's range to $[0, s - 2]$ means that the program only performs loop iterations for arrays of size $s \geq 2$. Consequently, it is trivially safe for the sizes 0 and 1. These are the sizes our bounded proof explores. Hence, BMC does not report any errors and leads us to wrongly believe that $trav^2_{0,2}$ is safe.

3 Completeness Thresholds

As illustrated above, bounded proofs are in general unsound approximations of unbounded proofs. A concrete approximation is sound iff we choose the bounds large enough, such that we can be sure that we do not miss any errors. We focus on bounding input sizes (in our examples array sizes), ignoring loop bounds that do not depend on these parameters.

Recall from § 1 that completeness thresholds are a concept from model checking of finite transition systems [20]. We borrow this terminology and apply it to memory safety verification. Hence, for a program $c(x)$ with input parameter $x \in X$, we call any subdomain $Q \subseteq X$ a *completeness threshold* (*CT*) for x in c if we can prove memory safety of c by only considering inputs from Q, i.e., $\forall x \in Q.\ \texttt{memsafe}(c(x)) \implies \forall x \in X.\ \texttt{memsafe}(c(x))$.

Intuitive CT Extraction. Returning to our example class of programs *Trav* implementing the array traversal pattern. Some of these are memory safe, some are not. So, let us try to compute completeness thresholds for these programs. First, let's take a look at the errors that might occur. This gives us an idea which sizes a sound bounded proof must cover. Any instance $trav^Z_{L,R}(a,s)$ iterates ascendingly $i = L, \ldots, s - R$ and accesses $a[i + Z]$. For sizes s that cause the ascending range $L, \ldots, s - R$ to be empty, we do not execute the loop at all. Any such run is trivially memory safe. Therefore, any meaningful bounded proof of $trav^Z_{L,R}(a,s)$ must include sizes s with $\{L, \ldots, s - R\} \neq \emptyset$, i.e., $s \geq L + R$.

Suppose $s \geq L + R$. An error occurs if the index $i + Z$ violates the array bounds, i.e., if $i+Z < 0$ or $i+Z \geq s$. Taking the index range into account, we see that we get an error if $L + Z < 0$ or $s - R + Z \geq s$ holds. We can simplify the latter to $Z - R \geq 0$.

Note that neither $L + Z < 0$ nor $Z - R \geq 0$ depend on the array size s. This means that as long as we focus on

sizes above the threshold $s \geq L + R$, the concrete choice of s does not influence whether an error occurs or not. In other words, it suffices for our bounded proof to only check a single (arbitrarily chosen) size $q \geq L + R$ and then we can extrapolate the result, i.e.,

$$\forall a.\, \texttt{memsafe}(trav^Z_{L,R}(a, q)) \implies \forall s. \forall a.\, \texttt{memsafe}(trav^Z_{L,R}(a, s))$$

Hence, any set $\{q\}$ for $q \geq L + R$ is a CT for the array size parameter s in $trav^Z_{L,R}$. We just found a uniform characterization of CTs for the entire class *Trav*. Note that $\{q\}$ is not necessarily the smallest CT. For safe instances such as $trav^0_{0,1}$, the empty set $\emptyset$ is a valid CT as well.

Our Approach. We study CTs for x in $c(x)$ by studying its *verification condition* (VC) . The latter is an automatically generated logical formula of the form $\forall x \in X.\ vc(x)$ and proving it entails memory safety of $c(x)$ for all choices of x. Next, we currently simplify $vc(x)$ by hand until it becomes clear how the choice of x affects the validity of $vc(x)$. Knowing this allows us to partition the domain into $X = \bigcup Q_i$. For each subdomain we get $vc_i(x) = \forall x \in Q_i.\ vc(x)$. If possible, we simplify each $vc'_i(x)$ into $vc''_i(x)$ based on the restricted subdomain Q_i with the goal to eliminate occurrences of x. If vc''_i does not mention x we pick any element of Q_i as representative Q'_i. Otherwise, $Q'_i = Q_i$. Hence, $\bigcup Q'_i$ is a CT for x in $c(x)$. In the following we elaborate this in more detail.

3.1 Approximating CTs via Verification Conditions

Now that we have an intuition for the CTs of *Trav*, let's turn our informal argument from above into a formal one. Formal definitions of the language and logic we consider and proofs for the presented lemmas can be found in the technical report [44].

Hoare Triples. We use Hoare triples [31] to express program specifications. A triple $\{A\}\ c\ \{B\}$ expresses that the following properties hold for every execution that starts in a state which satisfies precondition A: Firstly, the execution does not encounter any runtime errors. Secondly, it either (i) does not terminate or (ii) it terminates in a state complying with postcondition B.

In this work, we study the memory safety of programs that do not change the shape of the data structures they process. Hence, we choose preconditions that merely describe the memory layout of the data structures which our programs receive as input. For the array traversal program, we choose the predicate $\mathsf{array}(a, s)$ as precondition, which expresses that a points to a contiguous memory chunk of size s. Given that our target programs do not change the memory layout, specifications simplify to $\{A\}\ c\ \{A\}$. For the array traversal we get $\{\mathsf{array}(a, s)\}\ trav^Z_{L,R}\ \{\mathsf{array}(a, s)\}$.

Definition 3.1 (Completeness Thresholds for Programs)**.** Let $\{A\}\ c\ \{B\}$ be a program specification containing a free variable x with domain X. We call a subdomain $Q \subseteq X$ a *completeness threshold* for x in $\{A\}\ c\ \{B\}$ if

$$\models \forall x \in Q.\ \{A\}\ c\ \{B\} \quad \Rightarrow \quad \models \forall x \in X.\ \{A\}\ c\ \{B\}$$

We omit spelling out the pre- and postconditions when they are clear from the context. Instead we say that Q is a completeness threshold for x in program c.

Separation Logic. We use a first-order affine/intuitionistic separation logic with recursion predicates [40, 41, 45] to describe memory. Since we focus on heap-manipulating programs, we use assertions to describe heaps. Separation logic comes with a few special operators: (i) The points-to chunk $x \mapsto v$ describes a heap containing a location x which holds the value v. We write $x \mapsto \_$ to express that we do not care about the value stored in the heap cell. (ii) The separating conjunction $a_1 * a_2$ expresses that a_1 and a_2 describe disjoint heaps. Hence, $x \mapsto \_ * y \mapsto \_$ implies that $x \neq y$. (iii) The separating implication $a_1 \mathrel{-\!\!*} a_2$ can be read as a_2 *without* a_1. That is, combining the heap described by $a_1 \mathrel{-\!\!*} a_2$ with a disjoint heap described by a_1 yields a heap compliant with a_2. (iii) In our logic, the persistence modality $\square\, a$ means that a does not describe resources and hence holds under the empty heap (cf. [33]).

We assume that array denotes a (recursively defined) predicate, such that for every fixed size s, we can express it as iterated separating conjunction: $\mathsf{array}(a, s) \equiv \mathop{\Large *}_{0 \leq k < s} a[k] \mapsto \_$.

Verification Conditions. A common way to verify programs is via verification conditions [29, 43]. For any specification $\{A\}\ c\ \{B\}$, a *verification condition* (*VC*) is any logical formula vc, such that we can verify $\{A\}\ c\ \{B\}$ by proving vc, i.e., $\models vc \implies \models \{A\}\ c\ \{B\}$.

Definition 3.2 (Verification Condition)**.** We call an assertion a a *verification condition* for $\{A\}\ c\ \{B\}$ if

$$\models a \quad \Rightarrow \quad \models \{A\}\ c\ \{B\}.$$

Definition 3.3 (Completeness Thresholds for Assertions)**.** Let a be an assertion with a free variable x of domain X. We call a subdomain $Q \subseteq X$ a *completeness threshold* for x in a if

$$\models \forall x \in Q.\ a \quad \Rightarrow \quad \models \forall x \in X.\ a.$$

Consider a specification $\{A\}\ c\ \{B\}$ with a free variable $x \in X$ and a corresponding VC $\forall x \in X.\ vc$. Suppose we get a completeness threshold Q for x in vc. Knowing this threshold reduces correctness of the specification to the bounded VC, i.e., $\models \forall x \in Q.\ vc \Rightarrow \{A\}\ c\ \{B\}$. That is, we can derive unbounded guarantees from a bounded proof. We usually omit quantification domains when they are clear from the context.

Weakest Liberal Preconditions. A common way to generate VCs is via weakest liberal preconditions [24, 29]. For any program c and postcondition B, the *weakest liberal precondition* $\mathsf{wlp}(c, \lambda r.\ B)$ is an assertion for which

$$\forall A.\ \left(A \models \mathsf{wlp}(c, \lambda r.\ B) \implies \models \{A\}\ c\ \{B\} \right)$$

holds. That is, if the weakest liberal precondition holds for the starting state, then c does either not terminate or it terminates in a state complying with postcondition B. In particular, no memory error occurs during the execution. The canonical VC for $\{A\}\ c\ \{B\}$ is $\forall \overline{x}.\ A \rightarrow \mathsf{wlp}(c, \lambda r.\ B)$ where $\overline{x}$ is the tuple of variables occuring freely in A, c and B.

Limitations of CTs. In general, VCs are over-approximations. Hence, CTs derived from a VC do not always apply to the corrresponding program. Consider the specification $\{\mathsf{array}(a,s)\}\ trav^2_{0,2}\ \{\mathsf{array}(a,s)\}$ and any unsatisfiable assertion $vc_{\text{false}} \equiv \mathsf{False}$. Then $\forall s \in \mathbb{N}.\ vc_{\text{false}}$ is an over-approximating VC, since $\mathsf{False} \Rightarrow \models \{\mathsf{array}(a,s)\}\ trav^2_{0,2}\ \{\mathsf{array}(a,s)\}$. As discussed in § 2, a bounded proof that only covers sizes 0, 1 does not discover the errors in $trav^2_{0,2}$. Hence, the set $\{0\}$ is not a CT for x in $\{\mathsf{array}(a,s)\}\ trav^2_{0,2}\ \{\mathsf{array}(a,s)\}$. However, $\{0\}$ is a CT for s in vc_{false}, since $\forall s \in \{0\}.\ vc_{\text{false}} \equiv \mathsf{False}$ and $\mathsf{False} \Rightarrow \models \forall s \in \mathbb{N}.\ vc_{\text{false}}$. We see that we can only transfer a CT Q for some variable x derived from a VC vc to the corresponding program, if vc does not over-approximate with regard to x. (We are currently studying this connection.) Note that this is, however, the case for the examples we discuss in this paper. Though, even if vc does over-approximate, CT Q still applies to any proof considering a property at least as strong as vc.

As the name suggests, VCs derived from weakest preconditions yield very weak properties. It is reasonable to assume that often a bounded proof would imply a bounded version of the wlp-based VC. Therefore, it is reasonable to use them during our study of CTs.

Extracting CTs via VCs. Applying the above approach to our specification $\{\mathsf{array}(a,s)\}\ trav^Z_{L,R}(a,s)\ \{\mathsf{array}(a,s)\}$, we get a VC $\forall a.\ \forall s.\ vc_{\text{trav}}$, where vc_{trav} is as follows [2]:

$$
\begin{aligned}
vc_{\text{trav}} \ :=\ & \mathsf{array}(a,s) \rightarrow && \text{(\#1)}\\
& \mathsf{array}(a,s) && \text{(\#2)}\\
& * \ \square(\forall i.\ (L \le i \le s-R) \wedge \mathsf{array}(a,s) \mathrel{-\!\!*} && \text{(\#3)}\\
& \qquad \exists v.\ a[i+Z] \ \mapsto\ v \wedge \mathsf{array}(a,s))\\
& * \ (\mathsf{array}(a,s) \mathrel{-\!\!*} \mathsf{array}(a,s)) && \text{(\#4)}
\end{aligned}
$$

Here, (#1) is the precondition describing our memory layout. The separating conjunction (#2)-(#4) is the weakest precondition derived from our specification and vc_{trav} says that it should follow from precondition (#1).

The weakest precondition states that the memory layout must stay invariant under the loop execution. (#2) says that it should hold before the loop starts. (#3) demands that every loop iteration preserves the layout, i.e., that the layout description is a loop invariant. (#4) states this invariant implies the postcondition from our specification, which, again, is the unchanged memory layout.

Remember that we use an affine separation logic. Clearly, this VC contains many trivially obsolete parts. We can simplify vc_{trav} to vc_1:

$$
\begin{aligned}
vc_{\text{trav}} \ \equiv\ & \texttt{// Eliminate (\#2), (\#4)}\\
& \mathsf{array}(a,s) \rightarrow\\
& \quad \square(\forall i.\ (L \le i \le s-R) \wedge \mathsf{array}(a,s) \mathrel{-\!\!*}\\
& \qquad \exists v.\ a[i+Z] \ \mapsto\ v \wedge \mathsf{array}(a,s))\\
\equiv\ & \texttt{// Persistency makes pre. obsolete}\\
& \quad \square(\forall i.\ (L \le i \le s-R) \wedge \mathsf{array}(a,s) \mathrel{-\!\!*}\\
& \qquad \exists v.\ a[i+Z] \ \mapsto\ v \wedge \mathsf{array}(a,s))\\
\equiv\ & \texttt{// array(a,s) equiv. to } \mathop{\Large *}_{0\le k<s} a[k] \ \mapsto\ \_\\
& \quad \forall i.\ (L \le i \le s-R) \rightarrow (0 \le i+Z < s)\\
=:\ & vc_1
\end{aligned}
$$

This equivalent VC $\forall a.\ \forall s.\ vc_1$ does reflect the intuition we developed when analysing the program informally: For sizes $s < L+R$, the program does not perform any loop iterations and hence it is trivially memory safe. For bigger arrays, a memory error occurs iff index $i+Z$ violates the array bounds.

We can justify this intuition by partitioning the domain of s into $\mathbb{N} = \{0,\dots,L+(R-1)\} \cup \{L+R,\dots\}$. Let's analyse vc_1 for both subdomains separately. For a size $s_- < L+R$, we get

$$vc_1(s_-) \quad\equiv\quad \forall i.\ \mathsf{False} \rightarrow (0 \le i+Z < s_-) \quad\equiv\quad \mathsf{True}$$

So, we do not have to bother checking sizes $s < L+R$. For bigger sizes $s_+ \ge L+R$, we get

$$
\begin{aligned}
& vc_1(s_+) \ \equiv\ \forall i.\ (L \le i \le s_+ - R) \rightarrow (0 \le i+Z < s_+)\\
\equiv\ & \forall i.\ (L \le i \rightarrow 0 \le i+Z) \ \wedge\ (i \le s_+ - R \rightarrow i+Z < s_+)\\
\equiv\ & \forall i.\ (L \le i \rightarrow 0 \le i+Z) \ \wedge\ (i \le -R \rightarrow i+Z < 0)\\
=:\ & vc_2.
\end{aligned}
$$

Since s_+ does not occur freely in vc_2, the truth of $vc_1(s_+)$ does not depend on the choice of s_+. Remember that we have, $vc_{\text{trav}}(s_-) \equiv \mathsf{True}$ and $vc_{\text{trav}}(s_+) \equiv vc_2$. Hence,

$$\models \forall a.\ \forall s.\ vc_{\text{trav}} \quad\Leftrightarrow\quad \models \forall a.\ vc_{\text{trav}}(s_+)$$

We see that it suffices to check the original VC vc_{trav} for any size $s_+ \ge L-R$ to prove memory safety of our array traversing program $trav^Z_{L,R}$. That is, $\{s_+\}$ is a CT.

Characterizing CTs via Constraints. Note that the constraint $s \ge L-R$ we just derived is a uniform representation for the CTs of the entire class *Trav*. We often use constraints to concisely characterize CTs. A constraint formulates a property that is sufficiently strong so that every subdomain covering it is a CT. For every program $trav^Z_{L,R} \in \mathit{Trav}$, every subdomain $Q' \subseteq \mathbb{N}$ is a CT for the array size s if $Q' \cap \{s \in \mathbb{N} \mid s \ge L-R\} \neq \emptyset$.

[2] The weakest precondition calculus requires us to annotate loops with loop invariants. In the setting we study, the initial memory layout is invariant under the program's execution. The preconditions we consider describe exactly the initial memory layout, nothing else. Hence, we can reuse preconditions as loop invariants during the wlp computation.

```
for i in [L : s-R] do (
  n := a[i+Z];
  complex_fct(n, y, k)
)
```

$$=: \quad comp^{Z}_{L,R}(a, s, y, k)$$

$$M(a,s) \quad := \quad \mathsf{array}(a,s) * \mathsf{complex\_data}(y,k)$$
$$Comp \quad := \quad \{\{M\}\ comp^{Z}_{L,R}\ \{M\} \quad | \quad L,R,Z \in \mathbb{Z}\}$$

Figure 2. Class of programs involving a complex data structure and computation that do not depend on the array size s.

3.2 Modularity of Completeness Thresholds

Unrelated Data Structures. Consider the program $sum^{Z}_{L,R}(a, s, n)$ that attempts to sum up all elements of array a and writes the result to heap location n:

```
for i in [L : s-R] do
  !n := !n + !a[i+Z]
```

$$=: \quad sum^{Z}_{L,R}(a, s, n)$$

Analogously to $trav^{Z}_{L,R}$ it iterates through the array and attempts to read array elements. Additionally, it uses the read value to update the sum stored at heap location n. Let us assume that the array a and the result variable n do not alias. We get the specification $\{A\}\ sum^{Z}_{L,R}\ \{A\}$ for $A := \mathsf{array}(a,s) * n \mapsto \_$. We assume structured memory. Therefore, it is not possible to access n via an array access $a[...]$.

Intuitively, it is clear that the array size does not affect the memory accesses to heap location n. Hence, the CTs for s in $sum^{Z}_{L,R}$ should be the same as the ones for $trav^{Z}_{L,R}$. In fact, analysing the wlp-based VC for $sum^{Z}_{L,R}$ confirms this intuition. It has the form $\forall a.\forall s.\forall r.\ vc_{\text{sum}}$ and we can rewrite vc_{sum} into

$$vc_{\text{sum}} \quad \equiv \quad vc_{\text{trav}} * (n \mapsto \_ \to A)$$

where vc_{trav} is the VC from § 3.1 for the array traversing program $trav^{Z}_{L,R}$. Moreover, $\mathsf{freeVars}(A) = \{n\}$. Since s does not occur freely in $n \mapsto \_ \to A$, we can ignore it while searching for a CT for s in vc_{sum}.

Lemma 3.4 (VC Slicing)**.** *Let a, a_x, a_y be assertions with $x \in \mathsf{freeVars}(a_x)$ and $x \notin \mathsf{freeVars}(a_y)$ and $a \equiv a_x * a_y$. Let $Q \subseteq X$ be a CT for x in a_x. Then, Q is also a CT for x in a, i.e.,*

$$\models \forall x \in Q.\ \forall \overline{y} \in \overline{Y}.\ a \quad \Rightarrow \quad \models \forall x \in X.\ \forall \overline{y} \in \overline{Y}.\ a$$

We can extrapolate what we saw in the $sum^{Z}_{L,R}$ example to more complex classes of programs. Consider the class *Comp* presented in Fig. 2. A program $comp^{Z}_{L,R} \in Comp$ receives two non-aliasing data structures: an array a of size s and a complex data structure y of size k, described by the predicate $\mathsf{complex\_data}(y,k)$. $comp^{Z}_{L,R}$ reads elements from array a, stores the result in a local variable n and then calls a complex function `complex_fct(n,y,k)` which does neither depend on a nor s. The VC vc_{comp} will reflect this. That is, analogous to the example above, it should be expressible as $vc_{comp} \equiv vc_{\text{trav}} * (\mathsf{complex\_data}(y,k) \to \dots)$ where the right conjunct does not depend on s.

VC Slicing lemma 3.4 tells us that whenever we want to characterize a CT for a specific parameter, we can ignore all separated VC conjuncts that do not involve this parameter. Effectively, this means that we can ignore all the complex parts of $comp^{Z}_{L,R}$ that are not related to the array size while searching for a CT for s. This allows us to reduce the search to the CTs of *Trav*.

Compositionality. We can describe the CTs of complex programs in terms of the CTs of their building blocks. Consider the program $c_1; c_2$ and suppose that c_1 and c_2 are instances of patterns we studied before. So we know that each c_i corresponds to a VC $\forall x.\ vc_i$ with a CT Q_i for x. Let $vc_{1;2}$ be the VC for $c_1; c_2$ that we want to prove. Suppose it can be rewritten into $\forall x.\ vc_1 \wedge vc_2$. Then, we know that $Q_1 \cup Q_2$ is a CT for $vc_{1;2}$. Therefore, our approach to studying CTs is to study patterns and combinators.

Basic Patterns. We view basic patterns such as the array traversal pattern discussed above as the basic building blocks. They tend to occur frequently in programs and they are sufficiently concise to extract CTs by studying their VCs. In particular, we focus on traversal and access patterns that preserve the memory layout. For now, we focus on arrays, but we are going to generalize it to arbitrary inductive data structures.

Managing Complexity. One of our main goals is to describe CTs for interesting classes of programs. VCs tend to become very complex very fast as a program gets more complex. Hence, we need a way to deal with this complexity and to break the CT analysis down into simpler problems. Following the structure of the program we want to reason about is a natural approach.

Combinator Patterns. In order to exploit the program structure while analysing CTs, we need to study how control structures affect CTs. For instance, as described above, we can characterize the CT of a sequence $c_1; c_2$ as the union of the CTs derived from c_1 and c_2. Further, consider the command `if` e `then` c_1 `else` c_2 and suppose that we can describe CTs for a size s in each c_i via constraints $\bigwedge K_i$. Then, we can describe the CT for the entire command via the constraints $(e \wedge \bigwedge K_1) \ \vee \ (\neg e \wedge \bigwedge K_2)$.

4 Conclusion

Past approaches to program verification either targeted unbounded guarantees and relied on unbounded, often inductive, proofs or they targeted bounded guarantees and tried to approximate the program behaviour using techniques like bounded model checking. We have, however, seen little interaction between the two communities.

In this work we propose a new perspective on memory safety proofs that connects unbounded and bounded proofs. We show that we can reduce unbounded memory safety proofs to bounded ones for certain programs that traverse arrays and preserve the memory layout. For any such program considering a few select array sizes yields the same guarantees as considering arrays of all possible sizes. We call this concept *completeness thresholds* in reference to a similar concept from model checking of finite transition systems. Moreover, we show that studying verification conditions are an adequate way to study completeness thresholds.

5 Related Work

Completeness thresholds were first introduced by Kroening and Strichman [35]. So far, the study of CTs has been limited to finite state systems. Indeed, the well-known CTs for classes of LTL properties are defined with respect to the (recurrence) diameter of the finite state system in question (e.g. [7, 34]). Determining the worst-case execution time of a program and discovering upper bounds on loops by iterative unrolling can also be used to determine CTs [19, 26]. For a possibly infinite state system those CTs can naturally be infinite as well. By specializing in just one property, memory safety, we are able to characterize and possibly find useful CTs for these systems as well. Model checking for parameterized network topologies of identical (e.g. bisimilar [12] or isomorphic [28]) processes features a related concept to completeness thresholds called *cutoff*. That is, model-checking up to the cutoff implies correctness of scaling the topology up to infinitely many processes. Positive results exist for properties of such token rings [12, 28] but also other topologies [2, 22].

The model checking literature (cf. [18]) boasts a wealth of alternative approaches to obtain unbounded guarantees on finite state systems, e.g., k-induction [8, 46], Craig interpolation [38] and property-directed reachability [10, 27] with adaptions to the software verification setting (e.g. [5, 6, 25]).

Array-manipulating programs are well-studied across different domains [9, 11, 14, 32]. However, we consider our main contribution to be a novel approach to connect unbounded and bounded proofs about memory safety. Ultimately, as discussed in the outlook, we aim to generalize and automate our approach to tree-like data structures. In that regard, Mathur et al. [37] consider the special case of proving memory safety of heap-manipulating programs as well. They prove that memory safety is decidable if the initial heap is forest-like and the program only performs a single-pass over the datastructure (see also § 6). They do not cover arrays and buffer overflows, but they support (de-)allocation.

6 Outlook

CTs for Programs. Consider any program c with a free variable x and a corresponding VC vc. Suppose we derived a CT Q for x in vc. In general, this does not allow us to conclude that Q is also a CT for x in c. Intuitively, this is only true if vc does not over-approximate with regard to x. This holds for the programs and VCs studied in this paper. While CTs for VCs still tell us something about proofs targeting these VCs, our ultimate goal is to derive thresholds for programs. Hence, we are currently studying this connection.

Scalability. In this work, we focus on a restricted array traversal pattern to illustrate CTs. Our goal is to scale this approach to complex programs. Therefore, we are currently studying more array traversal and access patterns and combinators. This will allow us to better understand how the structure of programs affects the relation between bounded and unbounded proofs. Knowing this will allow us to characterise CTs for complex classes of programs that cover errors besides off-by-n errors.

Afterwards, we are going to extend our approach to include arbitrary inductive data types. In particular, we plan to describe CTs for a class of programs *Sort* that includes (safe and unsafe) implementations of in-place sorting algorithms involving nested loops. Once we managed that, we are going to investigate how allocation and deallocation affect CTs.

Decidability. The memory accesses we observe in sorting functions typically mainly depend on the size of the sorted data structure, the traversal strategy and a comparison relation $<$. Suppose we managed to derive finite CTs for the class *Sort*. We conjecture that this will be sufficient to conclude decidability of memory safety for *Sort*. Note that this targeted result would escape the scope of the related work by Mathur et al. [37]. The latter showed decidability of memory safety for a certain form of single-pass programs, i.e., programs that traverse a datastructure exactly once.

Improving BMC Guarantees. Bounded model checking suffers from the state space explosion problem [16, 17, 42] with respect to the chosen bounds. It is often only practical to check very small bounds on each parameter to keep the verification time practical. We plan to automate the approach introduced in this paper and to leverage existing static analysis techniques (e.g. [3, 36, 39, 47]) to simplify the generated VCs. Once we are able to automatically compute a small CT for one parameter, say the traversed data structure's size, we can adjust the corresponding bound. We can be sure that we didn't miss to check any size that could lead to memory errors. Hence, it would strengthen the guarantees we get from our bounded proof. Ultimately, we would like to integrate with the industrial-strength BMC tool CBMC [19, 21].

Further, it would speed up the verification time and free resources that can be spend on exploring other parts of the program. That is, the lowered size bound might allow us to increase other bounds that seem more important. We would be able to do this while keeping the verification time stable and without sacrificing guarantees.

References

[1] Mohammad Abdulaziz, Michael Norrish, and Charles Gretton. 2018. Formally Verified Algorithms for Upper-Bounding State Space Diameters. *Journal of Automated Reasoning* 61 (2018), 485–520.

[2] Benjamin Aminof, Swen Jacobs, Ayrat Khalimov, and Sasha Rubin. 2014. Parameterized Model Checking of Token-Passing Systems. In *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8318)*, Kenneth L. McMillan and Xavier Rival (Eds.). Springer, 262–281. https://doi.org/10.1007/978-3-642-54013-4_15

[3] Irina Mariuca Asavoae, Mihail Asavoae, and Adrián Riesco. 2018. Slicing from formal semantics: Chisel - a tool for generic program slicing. *Int. J. Softw. Tools Technol. Transf.* 20, 6 (2018), 739–769. https://doi.org/10.1007/s10009-018-0500-y

[4] Mohammad Awedh and F. Somenzi. 2004. Proving More Properties with Bounded Model Checking. In *CAV*.

[5] Dirk Beyer and Matthias Dangl. 2020. Software Verification with PDR: An Implementation of the State of the Art. In *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12078)*, Armin Biere and David Parker (Eds.). Springer, 3–21. https://doi.org/10.1007/978-3-030-45190-5_1

[6] Dirk Beyer, Nian-Ze Lee, and Philipp Wendler. 2022. Interpolation and SAT-Based Model Checking Revisited: Adoption to Software Verification. *CoRR* abs/2208.05046 (2022). https://doi.org/10.48550/arXiv.2208.05046 arXiv:2208.05046

[7] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*.

[8] Per Bjesse and Koen Claessen. 2000. SAT-Based Verification without State Space Traversal. In *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1954)*, Warren A. Hunt Jr. and Steven D. Johnson (Eds.). Springer, 372–389. https://doi.org/10.1007/3-540-40922-X_23

[9] Marius Bozga, Peter Habermehl, Radu Iosif, Filip Konecný, and Tomás Vojnar. 2009. Automatic Verification of Integer Array Programs. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5643)*, Ahmed Bouajjani and Oded Maler (Eds.). Springer, 157–172. https://doi.org/10.1007/978-3-642-02658-4_15

[10] Aaron R. Bradley. 2011. SAT-Based Model Checking without Unrolling. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6538)*, Ranjit Jhala and David A. Schmidt (Eds.). Springer, 70–87. https://doi.org/10.1007/978-3-642-18275-4_7

[11] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2006. What's Decidable About Arrays?. In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3855)*, E. Allen Emerson and Kedar S. Namjoshi (Eds.). Springer, 427–442. https://doi.org/10.1007/11609773_28

[12] Michael C. Browne, Edmund M. Clarke, and Orna Grumberg. 1989. Reasoning about Networks with Many Identical Finite State Processes. *Inf. Comput.* 81, 1 (1989), 13–31. https://doi.org/10.1016/0890-5401(89)90026-6

[13] Daniel Bundala, Joël Ouaknine, and James Worrell. 2012. On the Magnitude of Completeness Thresholds in Bounded Model Checking. *27th Annual IEEE Symposium on Logic in Computer Science* (2012), 155–164.

[14] Supratik Chakraborty, Ashutosh Gupta, and Divyesh Unadkat. 2020. Verifying Array Manipulating Programs with Full-Program Induction. In *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12078)*, Armin Biere and David Parker (Eds.). Springer, 22–39. https://doi.org/10.1007/978-3-030-45190-5_2

[15] Nathan Chong, Byron Cook, Konstantinos Kallas, Kareem Khazem, Felipe R. Monteiro, Daniel Schwartz-Narbonne, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. 2020. Code-Level Model Checking in the Software Development Workflow. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice* (Seoul, South Korea) *(ICSE-SEIP '20)*. Association for Computing Machinery, New York, NY, USA, 11–20. https://doi.org/10.1145/3377813.3381347

[16] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification*, E. Allen Emerson and Aravinda Prasad Sistla (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 154–169.

[17] Edmund M. Clarke. 2008. Model Checking – My 27-Year Quest to Overcome the State Explosion Problem. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Iliano Cervesato, Helmut Veith, and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 182–182.

[18] Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron A. Peled, and Helmut Veith. 2018. *Model checking, 2nd Edition.* MIT Press. https://mitpress.mit.edu/books/model-checking-second-edition

[19] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 2988)*, Kurt Jensen and Andreas Podelski (Eds.). Springer, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15

[20] Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. 2004. Completeness and Complexity of Bounded Model Checking. In *International Conference on Verification, Model Checking and Abstract Interpretation*.

[21] Edmund M. Clarke, Daniel Kroening, and Karen Yorav. 2003. Behavioral consistency of C and verilog programs using bounded model checking. In *Proceedings of the 40th Design Automation Conference, DAC 2003, Anaheim, CA, USA, June 2-6, 2003*. ACM, 368–371. https://doi.org/10.1145/775832.775928

[22] Edmund M. Clarke, Muralidhar Talupur, Tayssir Touili, and Helmut Veith. 2004. Verification by Network Decomposition. In *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3170)*, Philippa Gardner and Nobuko Yoshida (Eds.). Springer, 276–291. https://doi.org/10.1007/978-3-540-28644-8_18

[23] The MITRE Corporation. 2006. CWE-193: Off-by-one Error. https://cwe.mitre.org/data/definitions/193.html

[24] Edsger W. Dijkstra. 1976. A Discipline of Programming. Pentice Hall.

[25] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. 2011. Software Verification Using k-Induction. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6887)*, Eran Yahav (Ed.). Springer, 351–368. https://doi.org/10.1007/978-3-642-23702-7_26

[26] Vijay Victor D'Silva, Daniel Kroening, and Georg Weissenbacher. 2008. A Survey of Automated Techniques for Formal Software Verification. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 27, 7 (2008), 1165–1178. https://doi.org/10.1109/TCAD.2008.923410

[27] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. 2011. Efficient implementation of property directed reachability. In *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, Per Bjesse and Anna Slobodová (Eds.). FMCAD Inc., 125–134. http://dl.acm.org/citation.cfm?id=2157675

[28] E. Allen Emerson and Kedar S. Namjoshi. 1995. Reasoning about Rings. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 85–94. https://doi.org/10.1145/199448.199468

[29] Cormac Flanagan and James B. Saxe. 2001. Avoiding exponential explosion: generating compact verification conditions. In *ACM-SIGACT Symposium on Principles of Programming Languages*.

[30] Keijo Heljanko, Tommi A. Junttila, and Timo Latvala. 2005. Incremental and Complete Bounded Model Checking for Full PLTL. In *International Conference on Computer Aided Verification*.

[31] C. A. R. Hoare. 1968. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12 (1968), 576–580. https://doi.org/10.1145/363235.363259

[32] Ranjit Jhala and Kenneth L. McMillan. 2007. Array Abstractions from Proofs. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4590)*, Werner Damm and Holger Hermanns (Eds.). Springer, 193–206. https://doi.org/10.1007/978-3-540-73368-3_23

[33] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[34] Daniel Kroening, Joël Ouaknine, Ofer Strichman, Thomas Wahl, and James Worrell. 2011. Linear Completeness Thresholds for Bounded Model Checking. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 557–572. https://doi.org/10.1007/978-3-642-22110-1_44

[35] Daniel Kroening and Ofer Strichman. 2003. Efficient Computation of Recurrence Diameters. In *International Conference on Verification, Model Checking and Abstract Interpretation*.

[36] Isabella Mastroeni and Damiano Zanardini. 2008. Data dependencies and program slicing: from syntax to abstract semantics. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2008, San Francisco, California, USA, January 7-8, 2008*, Robert Glück and Oege de Moor (Eds.). ACM, 125–134. https://doi.org/10.1145/1328408.1328428

[37] Umang Mathur, Adithya Murali, Paul Krogmeier, P. Madhusudan, and Mahesh Viswanathan. 2020. Deciding memory safety for single-pass heap-manipulating programs. *Proc. ACM Program. Lang.* 4, POPL (2020), 35:1–35:29. https://doi.org/10.1145/3371103

[38] Kenneth L. McMillan. 2003. Interpolation and SAT-Based Model Checking. In *International Conference on Computer Aided Verification*.

[39] Louise E. Moser. 1990. Data Dependency Graphs for Ada Programs. *IEEE Trans. Software Eng.* 16, 5 (1990), 498–509. https://doi.org/10.1109/32.52773

[40] Peter W. O'Hearn. 2019. Separation logic. *Commun. ACM* 62, 2 (2019), 86–95. https://doi.org/10.1145/3211968

[41] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL*. https://doi.org/10.1007/3-540-44802-0_1

[42] D.Y.W. Park, U. Stern, J.U. Skakkebaek, and D.L. Dill. 2000. Java model checking. In *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*. 253–256. https://doi.org/10.1109/ASE.2000.873671

[43] Gaurav Parthasarathy, Peter Müller, and Alexander J. Summers. 2021. Formally Validating a Practical Verification Condition Generator. In *International Conference on Computer Aided Verification*.

[44] Tobias Reinhard. 2023. Completeness Thresholds for Memory Safety of Array Traversing Programs: Early Technical Report. arXiv:2211.11885 [cs.LO]

[45] John C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science* (2002), 55–74. https://doi.org/10.1109/LICS.2002.1029817

[46] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. 2000. Checking Safety Properties Using Induction and a SAT-Solver. In *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1954)*, Warren A. Hunt Jr. and Steven D. Johnson (Eds.). Springer, 108–125. https://doi.org/10.1007/3-540-40922-X_8

[47] Mark D. Weiser. 1984. Program Slicing. *IEEE Trans. Software Eng.* 10, 4 (1984), 352–357. https://doi.org/10.1109/TSE.1984.5010248

Received 2023-03-10; accepted 2023-04-21

Crosys: Cross Architectural Dynamic Analysis

Sangrok Lee
rexlee@nsr.re.kr
The Affiliated Institute of ETRI
Daejeon, Republic of Korea

Jieun Lee
jieunlee@nsr.re.kr
The Affiliated Institute of ETRI
Daejeon, Republic of Korea

Jaeyong Ko
bbxiix@nsr.re.kr
The Affiliated Institute of ETRI
Daejeon, Republic of Korea

Jaewoo Shim
tlawodn94@nsr.re.kr
The Affiliated Institute of ETRI
Daejeon, Republic of Korea

Abstract

Though there was a surge in the production of IoT devices, IoT malware analysis has remained a problem wanting for a clever solution. However, unlike PC and mobile, whose running environment is relatively standardized, IoT malware is rooted in Linux binary so that it can be built for various CPUs and with multiple libraries. For that, developing an effective dynamic analysis tool can be a challenging task.

In this paper, we present `Crosys`, a method for dynamic analysis of multi-architectural binaries in a single analysis host through intermediate language interpretation and binary rewriting. We explain how we elaborate binary lifting to assure both accuracy and stability. Then we propose cross-architectural dynamic analysis enabled by our work. In the end, we evaluated the stability of rewritten binary and the efficiency of dynamic analysis using technology.

CCS Concepts: • **Security and privacy** → **Malware and its mitigation**; • **Software and its engineering** → *Dynamic analysis*; • **Theory of computation** → *Program analysis.*

Keywords: cross architecture, dynamic analysis, malware, binary rewriting

ACM Reference Format:
Sangrok Lee, Jieun Lee, Jaeyong Ko, and Jaewoo Shim. 2023. Crosys: Cross Architectural Dynamic Analysis. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '23), June 17, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3589250.3596147

SOAP '23, June 17, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0170-2/23/06…$15.00
https://doi.org/10.1145/3589250.3596147

1 Introduction

We use many IoT devices in homes and offices, and the number of IoT devices is increasing by about 20% every year [40]. However, most IoT devices require a small form factor and operate with low power, so they have low computing power. Because of this, it is not easy to embed hardware and software modules for security.

Many security vulnerabilities have been discovered in IoT products, which have been used for cyber attacks[4]. In addition, since IoT devices connect to the network, if malware infects one device, it can instantly infect nearby IoT devices. Due to these characteristics of IoT, DDoS attacks exceeding 1 Tbps were possible through large-scale botnets [2]. However, IoT malware analysis is complex, and there are two notable reasons. The first one is the diversity of CPU architecture. IoT devices use various CPU architectures, including ARM, MIPS, x86, PowerPC, and SuperH [12, 28], in contrast with PCs and mobile; most PCs use one Intel CPU architecture, and mobile uses ARM architecture. The diversity of IoT device CPU architectures makes malware analysis an arduous task. For example, the Torii botnet discovered in 2018 targets more than 20 CPUs. Since each CPU has a different structure and instructions, it is almost impossible for an analyst to analyze binaries for more than 20 CPU architectures. Also, developing analysis tools to suit individual machine language is impractical.

Secondly, there is no standardized hardware structure in IoT, which can also be contrasted with PC and mobile devices with standardized device structures. To dynamically analyze a binary, it is required to build an execution environment and install an analysis tool. As an execution environment, an actual device and a virtual machine are standard options for analysts. When using a real device, obtaining it is not easy. Installing an analysis tool in commercial IoT products is difficult, and even if analysts succeeded in installing it, it might not be possible to open their terminals. In addition, due to the low computing power of IoT hardware, it is sometimes impossible to run tools such as dynamic binary instrumentation. When we decide to use a virtual machine as an execution environment, there is no standard virtual machine image in IoT due to its non-standardized structure.

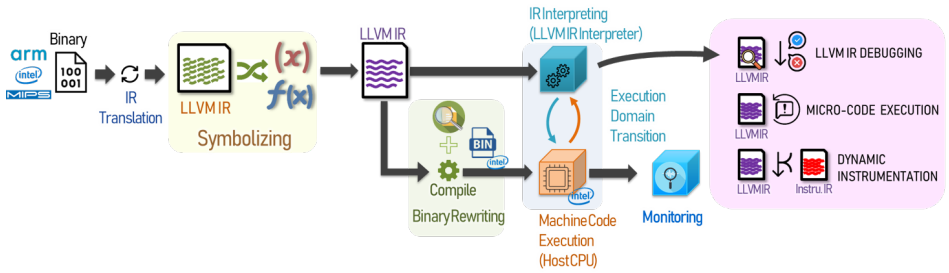


Figure 1. System overview

Even a virtual machine for the specific CPU may not exist. In other words, dynamic analysis may not be possible even though we can obtain a malicious binary.

In this situation, converting a binary into an intermediate language can be an appealing solution. Using intermediate language instead of the heterogeneous form of binaries, the analyst does not need to know each CPU's structure and assembly. Also, the analysis tool does not need to be developed targeting various CPUs since the analysis tool is now just for one language: the intermediate language. Moreover, in case of dynamic analysis, using intermediate language makes it possible to analyze binaries in a single execution environment rather than building an execution environment that is inevitably different for each IoT device. For these reasons, we present a method for dynamic analysis in a single analysis host environment through intermediate language interpretation and binary rewriting.

Many studies[8, 15, 35] suggested tools that analyze binary using intermediate language. However, these studies focused on accurately analyzing a specific CPU binary and executing a portion code of a binary file. We suggest a cross-architectural dynamic analysis method for IoT binary analysis. Rather than running and analyzing a specific code area, we translate the entire binary file into an intermediate language and analyze it dynamically on a single host. We summarize our contributions below:

- We suggest a novel way to analyze dynamically for multi-CPU binaries on a single host.
- We present implementation methods with an intermediate language interpreter and binary rewriting method.
- We evaluate `Crosys` on MIPS, x86-64, and ARM CPU binaries of Linux `coreutils`.
- We show that `Crosys` can accurately rewrite binaries and provide various analysis functions with fast execution performance.

2 Approach

`Crosys` has proposed a novel approach for analyzing IoT malware, which involves converting the malware binary into a host CPU executable, specifically an x86-64-based Linux executable. Following this conversion, dynamic analysis techniques are applied within the host environment to understand the malware's behavior better. In order to maintain the stability and accuracy of the converted binary during execution, it is crucial to employ effective methods. This section will outline the strategies utilized by `Crosys` to accomplish these goals.

2.1 Binary Lifting

The first step of `Crosys` is to convert the binary into LLVM IR code. It is crucial to extract the code accurately from the binary to obtain executable IR.

2.1.1 Linear Sweep Disassembly. `Crosys` employs the linear sweep disassembly algorithm[3, 33]. When all target addresses of branch instructions are known, the recursive traversal disassembly algorithm[3, 33] accurately extracts code sections from the binary. However, if an indirect branch's target address cannot be resolved, the disassembly process may stop, potentially causing code coverage issues. Moreover, data can be misidentified as code when the target address is statically but inaccurately calculated.

On the other hand, linear disassembly does not pose a code coverage issue. Instead, if data or compiler-generated padding exists between code, they can be disassembled as regular machine instructions. However, coverage is more crucial for execution stability. Even if we translate data or padding into IR code, the corresponding IR code will not execute because there won't be execution control flow for that code. Naturally, in the case of variable-length instructions like x86 or x86-64, misaligned instructions can be generated for code immediately adjacent to data. Nevertheless, instructions realign within 2-3 orders[3, 29]. We will address this in the following section.

2.1.2 QEMU *Tiny Code* to LLVM IR. Suppose we convert each CPU architecture machine language into LLVM IR individually. In that case, it will take a lot of effort and time, and there are many possibilities of inaccurate conversion[26]. Therefore, we use the method suggested in previous studies [14, 22] for getting LLVM IR code from binary(aka lifting). First, we convert binary to QEMU[7] *tiny code*, then convert

it to LLVM IR code. Since individual CPU instructions are converted to *tiny code* using QEMU, we only need to convert *tiny code* to LLVM IR. This strategy can significantly reduce the effort and time required for binary lifting and obtain accurate IR.

2.2 CPU Modeling for the Cross Architectural Execution

Modeling the CPU becomes even more critical when dealing with binary rewriting for cross-CPU architectures. For example, when translating an ARM binary to an x86-64 binary, differences in the number of registers and their sizes make a simple 1:1 mapping unfeasible.

The situation grows more complex with multiple source CPUs involved and the potential need to add new CPU architectures. Designing register mappings between source and target CPUs can be time-consuming and error-prone, demanding significant effort.

For this reason, `Crosys` avoids mapping source CPU registers to target CPU registers during binary translation. Instead, it represents the source CPU as a structured variable (i.e., virtual CPU) in the IR code, storing the results of IR instructions within the virtual CPU variable. Memory and IO operations remain unchanged, as they are on the target CPU system.

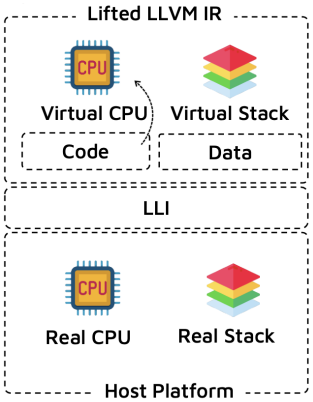


Figure 2. Modeling of Virtual CPU

The virtual CPU model sets itself apart from other binary translators[31, 34], as in those systems, source CPU registers are mapped to target CPU registers in the final binary. However, the virtual CPU approach offers the advantage of not necessitating design modifications to `Crosys` each time new CPU architectures emerge. Moreover, as we will discuss in the subsequent section, it plays a vital role in enabling execution domain transitions between the IR interpreter and the rewritten binary.

The downside of this approach is reduced execution speed compared to direct source-to-target register mapping. However, since `Crosys` serves as a prototype for cross-architectural dynamic analysis, performance improvement through direct mapping will be explored once a sufficient number of CPUs have been added to the system.

2.3 System Call Translation

There is a difference in the register size, endianness, and data alignment between CPU architectures. So, we have to convert the system call from the source CPU, where the original binary is executed, to that of the target CPU on which the IR code executes. For example, consider the `sys_read` system call function below.

```
long sys_read(unsigned int fd, char *buf, size_t
    count);
```

During the conversion from a 32-bit ARM to x86-64 when calling the `sys_read` function, we need to convert three input/output parameters: `char *`, `size_t`, and `long`. On 32-bit systems, these arguments have a length of 4 bytes, whereas on 64-bit CPUs, they are 8 bytes long. Additionally, if the source and target CPUs exhibit different endianness, a byte order swap becomes necessary. To address this in `Crosys`, we translate QEMU's system call conversion routine into LLVM IR code, which we then employ during IR interpretation or binary rewriting.

2.4 Symbolization for Execution and Analysis

In order to execute the original binary as an intermediate language, the IR code immediately after conversion from binary to IR is sufficient. However, the IR code at this stage has a low-level syntax similar to assembly code which does not have functions and variables. All data is stored directly in CPU registers and memory, and the location of data or IR code does not provide any meaning for binary analysis.

In order to quickly and accurately analyze the binary behavior, at least functions and variables related to registers or memory locations should be provided. In addition, in the case of system call calls, statically calculating the system call number and finding the system call function name through this number provides beneficial information for dynamic analysis.

2.4.1 Function Separation. The first step in function identification is to find the starting address of the function. For example, during the disassembly process, if the target address of a function call instruction, such as CALL of x86-64 or BL of ARM, is a constant, it can be seen as the function start address. `Crosys` also finds function start addresses in the IR code. In the case of a function call, the return address is stored in a specific location, and the value of the Program Counter Register is changed. At this time, the return address must be the value obtained by adding the length of the instruction to the current instruction address to be regarded as a function call. The location where the return address

is stored is the stack memory for x86-64, the lr register for ARM, and the v0 and v1 registers for MIPS. Therefore, if the program counter register value is changed after storing the return address in such a location on the intermediate language code, it is regarded as a function call, and the value stored in the program counter becomes the starting address of the called function.

The return from the callee function to the caller function uses both machine language and intermediate language code. When a machine language instruction such as ret is encountered during disassembly, it becomes the address of the function return instruction. Detection of function return instructions in the intermediate language is done in two steps. First of all, (i) the intermediate language code flow in which the Program Counter register is changed to a value stored in a register such as a stack or lr is a candidate first. (ii) Next, all stacks must be cleared. The stack pointer value at that location must have the value immediately before calling the Callee function. The position is regarded as a function return if both (i) and (ii) are satisfied.

Suppose both the function start and function return locations are detected. In that case, all codes (Basic Blocks) on the path from the function start address to the function return location can be seen as the function body code constituting the callee function.

2.4.2 System Call Recognition. System calls have separate machine instructions for each CPU. SYSCALL for x86-64 and MIPS CPU, SVC for ARM. Also, according to the calling convention [5, 30, 32, 41], each CPU's system call number register is fixed. x86-64 has rax, MIPS has v0, and ARM embeds the number in its SVC instruction. We can retrieve the system call number through a simple data flow analysis. And we can extract the system call's function type through the kernel source. So, we can convert the system call instruction into a system call function.

2.5 Ensure Error Resistance for Execution Stability

2.5.1 Indirect Branch. If there is an indirect branch instruction in the binary, the IR code (function or basic block) corresponding to the target address must exist. This is because, in LLVM IR, it cannot be the target of a branch instruction other than the start of a function or basic block. In previous studies[6, 14], a method for statically calculating the target address of an indirect branch was presented, but complete accuracy was not possible. `Crosys` solves this problem by setting up a look-up table [34, 36] that converts functions with indirect branches to target addresses → Basic Block addresses for execution stability.

2.5.2 Incomplete Function Separation. The previous section discussed the symbolization process, noting that accurate symbol recovery without errors can be challenging [25]. In cases of function recovery, certain code regions might not be recognized as a function, or an indirect branch target could exist outside of it. `Crosys` addresses this issue by creating a significant "umbrella function" that encompasses the entire binary code [20, 34]. Similar to a prior study [20], the umbrella function is utilized when an indirect call or branch instruction lacks a target due to incomplete function separation.

In the previous study [20], the LLVM invoke instruction was used for calling functions rather than the call instruction. If an error occurred within the called function, the process would return to the invoke code point with stack rewinding and execute the alternative umbrella function code with the help of invoke. However, `Crosys` employs the LLVM tail call instruction for function calls, enabling the corresponding umbrella function code to be executed without reverting to the code prior to the call, even if an error emerges within the function.

Additionally, `Crosys` generates a straightforward lookup table for the umbrella function, mapping each machine instruction to a single IR basic block. This approach may increase code size but ensures that indirect calls and branches are consistently resolved [34].

2.5.3 Incomplete Data Symbols. Similar to previous studies [20, 34, 36], `Crosys` loads the original binary's data into memory, allowing the IR code to access the data even when symbols are not recovered. This approach enhances execution stability by avoiding the use of partially recovered symbol variables in the IR code. Nevertheless, during instrumentation or debugging, `Crosys` supplies the restored data symbol as supplementary information.

2.5.4 Shadow Basic Blocks. Variable-length machine instructions, like those in x86-64, can begin from nearly any point, which means that data or padding might be identified as regular machine instructions as mentioned in the previous section. In such cases, the IR basic block comprises misaligned machine instructions. During control flow analysis, we may come across a direct branch instruction pointing to the middle of a machine instruction.

Rather than splitting the misaligned IR basic block, we introduced the concept of a shadow block, in which `Crosys` creates a new basic block starting from the precise target address. Consequently, the existing and newly created basic blocks overlap in the code address space to some extent.

The shadow block strategy is also advantageous for machine instructions with prefixes, like LOCK in x86 and x86-64 architectures, since both the prefix and the original instructions can serve as branch targets in such cases.

3 Cross Architectural Dynamic Analysis

After converting executable LLVM IR code from a binary file, the IR code can be executed and analyzed in three ways.

3.1 Interpreting Based Dynamic Analysis

The first is interpreting the IR code. `Crosys` uses the LLVM interpreter (aka LLI)[1] included in the LLVM Framework for this. In a previous study, we published LLVM IR Debugger by implementing debugging commands including breakpoint, step, and data watch in this LLI[23]. In addition to the debugging, `Crosys` provides more useful analysis functions:

1. **Micro Code Execution** selectively executes a specific portion of the code.
2. **Forced Execution** directly executes the code at a specific memory address.
3. **Code Instrumentation** integrates analysis code before and after functions, basic blocks, and instructions.

These can be used as a dynamic instrumentation tool. Interpreter-based analysis is the most versatile and powerful of the three analysis methods.

3.2 Binary Rewriting Based Dynamic Analysis

Binary rewriting statically converts ARM, MIPS, and x86-64 binaries into an LLVM IR code, then compiles it to the x86-64 binary. We may insert the IR code during compilation to monitor the binary's execution or change some data. A newly created x86-64 binary can output the behavior while executing the x86-64 host. Dynamic analysis based on binary rewriting is effective for analyzing malware that detects the presence of an analysis tool because it does not install any tools outside itself.

3.3 Hybrid Analysis

LLI showed a performance degradation of at least tens to thousands of times compared to the native binary execution speed. For interpreting and rewriting binary, we use the same LLVM IR code. That is, execution of the rewritten binary and IR interpretation is possible with the 'same' IR. Analyzing malware with an IR interpreter has a tremendous advantage because we can apply various IR optimizations. So If we execute only important code parts with the interpreter and the other code parts with binary in conjunction, we can achieve both execution performance and analysis efficiency.

So, `Crosys` transitions the execution domain between the interpreter and the rewritten binary. To seamlessly shift execution between the interpreter and binary, it is crucial to synchronize the execution context between them, meaning we need to preserve the execution results of the previous domain in the subsequent part. The virtual CPU serves as the key execution context in this regard. Represented as a structure variable in IR and a global data in the rewritten binary, the virtual CPU allows both the interpreter and the binary to access the same memory region. Consequently, the interpreter and binary update their execution results to the same context, that is, the virtual CPU. We published the hybrid analysis method in 2022[24].

4 Evaluation

In order to provide better experience in IoT malware dynamic analysis, it is important to provide various dynamic analysis functionalities while assuring stability, accuracy and efficiency of the analysis tool. Therefore, we conducted two experiments of `Crosys` . In the first experiment we measured the execution accuracy of the rewritten binary to see if `Crosys` was suitable to be used as dynamic analysis tool. In the second experiment, we show the efficiency of our hybrid analysis by measuring the running time with/without transition between interpreter and rewritten binary.

4.1 Experiment 1: Correctness

Ensuring the generation of functional IR code is vital for the effectiveness of cross-architectural analysis. IoT malware bears similarities to Linux `coreutils` in the absence of a GUI and the presence of numerous IO operations, including file system actions. In fact, Linux `coreutils` binaries are even more complex in some aspects of program structure, as they frequently utilize low-level features like inode access. Due to these factors, We assessed the execution accuracy of `Crosys` using Linux `coreutils` binaries as test samples

Initially, we compiled the `coreutils` source code to produce x86-64, ARM, and MIPS binaries, which we then translated into x86-64 binaries. For the C libraries, we utilized glibc for x86-64, musl for MIPS, and uclibc for ARM. Since the `coreutils` incorporates the GNU automake test framework [19], we can verify if the compiled coreutils binaries execute without errors. We also employed this `coreutils` test suite to evaluate the execution success rate of the rewritten binaries. The experiment involved 107 binaries and roughly 610 tests.

Table 1. `coreutils` test results of the rewritten binaries. When a test case relies on non-portable tools, prerequisites, or a specific system, and these dependencies are not adequately met, the test case is skipped(**Skip**) and excluded from the success rate calculation[18].

CPU Architecture	Pass	Fail	Skip	Success rate
MIPS	431	32	127	93.1%
x86-64	439	43	135	91.1%
ARM	322	64	182	83.4%

Table 1 shows the `coreutils` test suite results for each CPU. The success rate of the rewritten binary shows 93%, 91%, and 83% for each CPU of MIPS/x86-64/ARM.

The failure mainly occurred due to missing multi-thread support and some unsupported system call conversions. Multi-thread support in the IR and some system call conversions are not yet implemented. `Crosys` is our first dynamic analysis prototype, and the missing functionalities could be solved by engineering work.

4.2 Experiment 2: Execution Transition Between the Interpreter and the Rewritten Binary

Crosys offers a variety of analysis features, including debugging, forced execution, microcode execution, and instrumentation. Furthermore, by compiling IR to x86-64 host binaries, fast execution is achievable. Crosys enables intensive analysis through an interpreter while quickly executing non-critical or known code, such as libraries, using the rewritten binary. This execution transition allows for optimal analysis efficiency, balancing accuracy and speed.

To evaluate the stability and performance enhancements of execution domain conversion, we utilized 78 ARM coreutils binaries. First, we converted each ARM binary to LLVM IR and rewrote it as an x86-64 binary. To test the stability of the execution transition, we executed the main function using the IR interpreter and the remaining code with the rewritten binary, and vice versa. Consequently, all 78 samples demonstrated stable execution domain transitions.

To evaluate the speed enhancement, the binary executed the main function from its entry point until it returned, while the interpreter managed the remaining code. Table 2 showcases the execution time difference for some samples when using only the interpreter versus transitioning between the binary and interpreter. A comprehensive comparison of all samples is available in Appendix A. The results demonstrate a reduction in execution time from 5% to 99%.

Table 2. Reducing execution time via transition: **I** denotes interpreter-only execution time, while **IB** indicates the time after applying execution switching. **Speedup** represents the decreased execution time rate of **IB** compared to **I**.

Program Name	I (sec.)	IB (sec.)	Speedup
TRUE	0.46	0.43	5.96%
rmdir	0.73	0.65	10.68%
ln	0.87	0.72	17.06%
tail	1.84	0.53	71.02%
paste	1.77	0.51	71.08%
fmt	1.84	0.52	71.65%
ls	44.87	0.57	98.72%
du	64.32	0.56	99.14%
df	492.35	0.73	99.85%

Upon examining the test results, we can see that binaries using a significant amount of IO, such as accessing the file system, exhibit greater improvements in execution speed.

5 Limitations and Future Work

Crosys can analyze statically linked binaries. According to a recent study[12], more than 80% of IoT malware was statically linked. However, support for dynamically linked binaries are required for Crosys to be utilized more. To execute the dynamically linked binaries on a single host, we must be able to give the same features as the original libraries (ARM, MIPS) on the x86-64 host. That is a big challenge, and we are working on solving this problem.

IoT malware focused on launching DDoS attacks aims to infect a broad spectrum of hardware devices by targeting multiple CPUs, typically using common actions to achieve this. However, in situations like VPNfilter[27], where specific peripherals are targeted for destruction, emulating the hardware becomes essential. We believe that even without investigating the firmware emulation mentioned in previous studies[9–11, 37], examining peripheral simulations provided by the host continues to be a critical area of research.

In Hybrid Analysis, domain transitions can only occur at function entry points, making mid-function changes impossible. To enhance performance and analysis efficiency, we intend to explore the On-Stack Replacement (OSR) approach [13, 16, 21].

6 Related Work

SecondWrite[36] and LLBT[34] convert the entire binary file into IR code and then compile it to create target binaries. They use a lookup table similar to our Crosys for handling an indirect branch. Crosys differs from these studies in that we use IR interpreting and binary rewriting for multiple CPU architectures' binaries. Crosys is practical for binary analysis because it can provide powerful analysis features through the interpreter and high speed through the rewritten binary.

In static binary rewriting, "reassembly"[17, 38, 39] is being explored, utilizing enhanced symbol-recovery instead of lookup tables. However, it still cannot ensure execution stability due to symbolization errors.

Previous studies[9, 11, 37] proposed firmware emulation for generating execution environments when IoT devices are unavailable. In contrast, CROSYS diverges from these by converting the target binary to the host one, running it within a host execution environment.

7 Conclusion

This paper presents a unique cross-architectural dynamic analysis tool for binary of various CPU architectures. Through Crosys, execution and analysis are possible on a single x86-64 host without building a dynamic analysis environment for each CPU. First, we present an approach to guarantee execution stability. We use linear sweep disassembly and many mitigation methods for various symbolization errors. Next, through interpreting and binary rewriting, Crosys can provide many analysis features, including debugging, microcode execution, and dynamic instrumentation. In particular, we show that the IR interpreter and the rewritten binary execution transition can ensure analysis accuracy and performance.

A Execution Time Reduction through Execution Transition. All Samples

Program Name	I (sec.)	IB (sec.)	Speedup
TRUE	0.461719	0.434187	5.96%
FALSE	0.55844	0.445303	20.26%
test	0.625229	0.511884	18.13%
rmdir	0.725612	0.648099	10.68%
unlink	0.727512	0.512901	29.50%
readlink	0.728667	0.501911	31.12%
echo	0.737979	0.518864	29.69%
sync	0.740831	0.516127	30.33%
nice	0.741142	0.509698	31.23%
link	0.74187	0.526826	28.99%
dirname	0.76135	0.564253	25.89%
printf	0.768333	0.526555	31.47%
truncate	0.776254	0.510504	34.23%
mkdir	0.78942	0.508165	35.63%
touch	0.797387	0.522942	34.42%
basename	0.799523	0.513162	35.82%
sleep	0.801026	0.541118	32.45%
uname	0.804692	0.526262	34.60%
cat	0.817284	0.593379	27.40%
seq	0.825189	0.511075	38.07%
ln	0.867054	0.719113	17.06%
expr	0.875859	0.527643	39.76%
factor	0.892823	0.526694	41.01%
realpath	0.897458	0.512466	42.90%
head	0.949342	0.541603	42.95%
chmod	0.966465	0.513321	46.89%
rm	1.024786	0.557438	45.60%
logname	1.028615	0.656174	36.21%
users	1.031201	0.540084	47.63%
nproc	1.073188	0.515773	51.94%
tty	1.107171	0.515606	53.43%
pwd	1.162965	0.525917	54.78%
cp	1.289204	0.565434	56.14%
base64	1.366554	0.523859	61.67%
base32	1.509979	0.510947	66.16%
comm	1.510527	0.530421	64.89%
fold	1.523586	0.547396	64.07%
cksum	1.563944	0.511537	67.29%
paste	1.771677	0.512436	71.08%
expand	1.791237	0.553407	69.10%
fmt	1.835495	0.520367	71.65%
tail	1.843137	0.534065	71.02%
nl	2.084801	0.657738	68.45%
make-prime-list	2.246213	0.425282	81.07%
split	2.379756	0.655259	72.47%
join	2.70798	0.544395	79.90%
hostid	2.737518	0.516916	81.12%
date	2.847946	0.539231	81.07%
md5sum	2.999575	0.511118	82.96%
tac	3.003981	0.556339	81.48%
sha1sum	3.642144	0.511669	85.95%
groups	4.219185	0.531426	87.40%
chgrp	4.330213	0.55672	87.14%
sha224sum	4.920892	0.518509	89.46%
pr	5.057979	0.518662	89.75%
sha256sum	5.458243	0.520081	90.47%
whoami	5.747042	0.543458	90.54%
chown	6.040226	0.550252	90.89%
pinky	6.108807	0.530833	91.31%
shuf	7.364342	0.50692	93.12%
uniq	9.083064	0.521573	94.26%
sum	9.437394	0.512638	94.57%
b2sum	10.77085	0.520908	95.16%
printenv	12.38841	0.516259	95.83%
id	13.41932	0.543351	95.95%
env	13.64609	0.571996	95.81%
stat	16.26867	0.527917	96.76%
basenc	17.54599	0.527348	96.99%
dircolors	17.6935	0.547339	96.91%
unexpand	19.97533	0.550394	97.24%
shred	21.79564	0.619357	97.16%
tsort	21.96775	0.519993	97.63%
od	29.9838	0.51249	98.29%
dir	37.74081	0.559946	98.52%
ls	44.871	0.573	98.72%
du	64.32186	0.556205	99.14%
vdir	152.6706	0.661431	99.57%
df	492.351	0.726253	99.85%

References

[1] 2022. *lli - directly execute programs from LLVM bitcode.* https://llvm.org/docs/CommandGuide/lli.html

[2] Cybersecurity & Infrastructure Security Agency. 2017. *Heightened DDoS Threat Posed by Mirai and Other Botnets.* Retrieved October 17, 2017 from https://www.cisa.gov/news-events/alerts/2016/10/14/heightened-ddos-threat-posed-mirai-and-other-botnets

[3] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries.. In *USENIX Security Symposium*. 583–600.

[4] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. 2017. Understanding the mirai botnet. In *26th {USENIX} security symposium ({USENIX} Security 17)*. 1093–1110.

[5] arm. 2023. *ABI for the Arm 32-bit Architecture.* https://developer.arm.com/Architectures/ApplicationBinaryInterface

[6] Gogul Balakrishnan and Thomas Reps. 2004. Analyzing memory accesses in x86 executables. In *Compiler Construction: 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29-April 2, 2004. Proceedings 13*. Springer, 5–23.

[7] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX annual technical conference, FREENIX Track*, Vol. 41. California, USA, 46.

[8] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. 2011. BAP: A binary analysis platform. In *Computer Aided*

Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23. Springer, 463–469.

[9] Daming Dominic Chen, Manuel Egele, Maverick Woo, and David Brumley. 2016. Towards fully automated dynamic analysis for embedded firmware. In *Proc. of NDSS*. 21–24.

[10] Abraham Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. Halucinator: Firmware re-hosting through abstraction layer emulation. In *Proceedings of the 29th USENIX Security Symposium*.

[11] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. 2016. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. 437–448.

[12] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. 2018. Understanding linux malware. In *2018 IEEE symposium on security and privacy (SP)*. IEEE, 161–175.

[13] Daniele Cono D'Elia and Camil Demetrescu. 2016. Flexible on-stack replacement in LLVM. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. 250–260.

[14] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. rev. ng: a unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*. 131–141.

[15] Adel Djoudi and Sébastien Bardin. 2015. Binsec: Binary code analysis with low-level regions. In *Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 21*. Springer, 212–217.

[16] Stephen J Fink and Feng Qian. 2003. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE, 241–252.

[17] Antonio Flores-Montoya and Eric Schulte. 2020. Datalog disassembly. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 1075–1092.

[18] Free Software Foundation. 2021. *GNU Automake - Generalities about Testing*. Retrieved October 1, 2021 from https://www.gnu.org/software/automake/manual/html_node/Generalities-about-Testing.html

[19] Free Software Foundation. 2021. *GNU Automake - Support for test suites*. Retrieved October 1, 2021 from https://www.gnu.org/software/automake/manual/html_node/Tests.html

[20] Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, Giovanni Agosta, et al. 2019. Performance, correctness, exceptions: Pick three. In *Binary Analysis Research Workshop*.

[21] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*. 32–43.

[22] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. 2012. HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. 104–113.

[23] Hyukmin Kwon Jaewoo Shim and Sangrok Lee. 2020. *A Cross Debugger for Multi-Architecture Binaries*. Retrieved April 6-7, 2020 from https://llvm.org/devmtg/2020-04/talks.html

[24] Jieun Lee Jaeyong Ko, Sangrok Lee and Jaewoo Shim. 2022. *Execution Domain Transition: Binary and LLVM IR can run in conjunction*. Retrieved November 8-9, 2022 from https://www.youtube.com/watch?v=s7nNYZvkGi8

[25] Hyungseok Kim, Soomin Kim, Junoh Lee, Kangkook Jee, and Sang Kil Cha. 2023. Reassembly is Hard: A Reflection on Challenges and Strategies. (2023).

[26] Soomin Kim, Markus Faerevaag, Minkyu Jung, Seungll Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. 2017. Testing intermediate representations for binary analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 353–364.

[27] William Largent. 2018. *New VPNFilter malware targets at least 500K networking devices worldwide*. Retrieved May 23, 2018 from https://blog.talosintelligence.com/vpnfilter/

[28] Yen-Ting Lee, Tao Ban, Tzu-Ling Wan, Shin-Ming Cheng, Ryoichi Isawa, Takeshi Takahashi, and Daisuke Inoue. 2020. Cross platform IoT-malware family classification based on printable strings. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 775–784.

[29] Cullen Linn and Saumya Debray. 2003. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*. 290–299.

[30] Hongjiu Lu, Michael Matz, Milind Girkar, Jan Hubiaka, Andreas Jaeger, and Mark Mitchell. 2018. System v application binary interface amd64 architecture processor supplement (with lp64 and ilp32 programming models) version 1.0.

[31] Koh M. Nakagawa. 2021. *Reverse-engineering Rosetta 2 part1: Analyzing AOT files and the Rosetta 2 runtime*. Retrieved February 19, 2021 from https://ffri.github.io/ProjectChampollion/part1/

[32] UNIX PRESS. 1996. System V application binary interface: MIPS Architecture Processor Supplement.

[33] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. 2002. Disassembly of executable code revisited. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* IEEE, 45–54.

[34] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. 2012. LLBT: an LLVM-based static binary translator. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*. 51–60.

[35] Y Shoshitaishvili, R Wang, and Ch Salls. 2016. The Art of War: Offensive Techniques in Binary Analysis", IEEE Symposium on Security and Privacy. (2016).

[36] Matthew Smithson, Khaled ElWazeer, Kapil Anand, Aparna Kotha, and Rajeev Barua. 2013. Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 52–61.

[37] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard Shrobe, and Mathias Payer. 2019. Firmfuzz: Automated iot firmware introspection and analysis. In *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*. 15–21.

[38] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again.. In *NDSS*.

[39] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable disassembling. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 627–642.

[40] Philipp Wegner. 2023. *Global IoT market size to grow 19% in 2023 —IoT shows resilience despite economic downturn*. Retrieved February 7, 2023 from https://iot-analytics.com/iot-market-size/

[41] Steve Zucker and Kari Karhi. 1995. System V Application Binary Interface: PowerPC™ Processor Supplement.

Received 2023-03-10; accepted 2023-04-21

RaceInjector: Injecting Races to Evaluate and Learn Dynamic Race Detection Algorithms

Michael Wang
mi27950@mit.edu
CSAIL, MIT
USA

Shashank Srikant
shash@mit.edu
CSAIL, MIT,
MIT-IBM Watson AI Lab
USA

Malavika Samak
malavika@csail.mit.edu
CSAIL, MIT
USA

Una-May O'Reilly
unamay@csail.mit.edu
CSAIL, MIT,
MIT-IBM Watson AI Lab
USA

Abstract

There exist no sound, scalable methods to assemble comprehensive datasets of concurrent programs annotated with data races. As a consequence, it is unclear how well the multiple heuristics and SMT-based algorithms, that have been proposed over the last three decades to detect data races, perform. To address this problem, we propose RaceInjector—an SMT-based approach which, for any given program, creates arbitrarily many program traces of it containing injected data races. The injected races are guaranteed to follow the given program's semantics. RaceInjector hence can produce an arbitrarily large, labeled benchmark which is independent of how detection algorithms work. We demonstrate RaceInjector by injecting races into popular program benchmarks and generating a small dataset of traces with races in them. Among the traces RaceInjector generates, we begin to find counterexamples which four state-of-the-art race detection algorithms fail to detect. We thus demonstrate the utility of generating such datasets, and recommend using them to train machine learning-based models which can potentially replace and improve upon existing race-detection heuristics.

CCS Concepts: • **Software and its engineering → Concurrent programming languages**; **Parallel programming languages**.

Keywords: Dynamic race detection algorithms, Race injection, Dataset generation, SMT-solvers

SOAP '23, June 17, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0170-2/23/06.
https://doi.org/10.1145/3589250.3596142

ACM Reference Format:
Michael Wang, Shashank Srikant, Malavika Samak, and Una-May O'Reilly. 2023. RaceInjector: Injecting Races to Evaluate and Learn Dynamic Race Detection Algorithms. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '23), June 17, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3589250.3596142

1 Introduction

Data race detection in concurrent programs using their execution traces, *i.e.* dynamic analysis, has been shown to be in NP-hard [23]. Practical algorithms designed to detect data races hence rely either on heuristics [17, 19, 22, 24, 28, 29] or SMT-solvers [7, 11, 16, 26, 32]. The goal of these algorithms is to start with a trace, and determine if two conflicting accesses in different threads to a shared variable can occur concurrently in an alternate execution of the program. Despite numerous such algorithms having been proposed over the last few decades, it is surprising that there exist no comprehensive benchmarks comprising industry-grade software projects which have races annotated in them—either annotated in the source code or in the traces—which would help rigorously evaluate these algorithms. Consequently, it is unclear what the true classification accuracy rates (true-positive, true-negative, *etc.*) of these algorithms are. For instance, none of the larger benchmark datasets such as DaCapo [2], which have all been repeatedly and extensively used in the evaluation of multiple race detection algorithms cited above, have any ground-truth annotations.

One key reason for the lack of such datasets is the absence of sound, scalable methods to assemble them. A few prior works [9, 13, 20, 33] have released expert-annotated datasets containing races. However, they are too small to be effective. Jacontebe [20] contains a total of 19 data race bugs, RadBench has 10 bugs, while GoBench [33] has 103 bugs for the Go language. With such small datasets, it is hard to evaluate if current algorithms commonly miss any race patterns.

Another approach to assembling such datasets has been to run SMT-based race detection algorithms on industry-grade software projects [9]. SMT-based race detection approaches, by their design, can provably detect true-positive data races. The detected true-positive data races are used as annotations and released as datasets. A major limitation of these approaches however is the relatively small number of constraints that SMT-solvers can solve at once. Longer, more complex real-world software produce longer execution traces, which in turn non-linearly increase the number of constraints which SMT-solvers have to solve. Detection algorithms typically circumvent this limitation by breaking the trace into fixed-length windows and solving each window as if they are independent of others [12]. The assumption of independence of windows is not practical, thus limiting the number and quality of races that can be detected. Moreover, assembling a dataset using data races detected by known detection approaches limits the nature of races we can test other detection algorithms on. We further discuss other prior works in Section 5.

As a direct consequence of this absence of scalable methods to assemble comprehensive annotated datasets, we argue that the metrics that have been employed to evaluate any new race detection algorithm do not accurately represent their performance. Further, we argue that the lack of such annotated datasets has potentially stifled the state-of-the-art.

• **Evaluating race detection algorithms.** Presently, the efficacy of newly proposed race detection algorithms is primarily measured by the increase in the number of identified race conditions when compared to a previous algorithm. Among heuristics-based algorithms [17, 19, 22, 24, 28, 29], each new algorithm has successively proposed a set of rules which purportedly improves upon previous algorithms. The newer algorithms then demonstrate detecting races which the previous algorithms did not. The newly detected races are verified manually by experts. In this process, it is unclear how many true-positives and false-negatives each new set of rules introduce. When an algorithm reports finding N (say) new bugs which a previous algorithm had not found, it is unclear what N is relative to—the total number of bugs which either of these algorithms are supposed to find in the first place. With surprisingly little attention paid to this essential metric, it is unclear what the state of progress in data race detection algorithms has been over the years.

Further, it is unclear whether an improvement proposed using a set of new rules results in detecting newer *classes* of data races, assuming there exist multiple semantic classes of use-cases in a program's execution behavior which manifest as race bugs. It is quite possible that a proposed improvement to an existing algorithm, while detecting a few new bugs, may not necessarily cover a significantly larger set of such semantic classes. An annotated dataset with a diverse set of bugs in them is the first step in establishing and quantifying the semantic classes a detection algorithm covers.

• **Training ML models.** We posit that it is possible for data-driven methods to replace the many heuristics which have been proposed over the years for race detection. Heuristics for race detection involve learning to draw edges between events of interest in a program's execution trace, and identifying cliques of connected or disconnected events. These cliques are then used to reason about and infer the existence of potential races. These heuristics typically guarantee soundness only for the first race they detect. Given the inadequacy of existing guarantees, machine learning (ML) models provide a practical alternative. ML models have been shown to outperform expert-crafted heuristics when reasoning about graph-based data in the domains of compiler optimization [5], network-graph analysis [3], pointer analysis [14], fault localization [21] and more. While such learned models will not be able to guarantee soundness even for the first detected race, they may well offer an improved performance in reasoning with trace-based information over expert-crafted heuristics. However, a key requirement to train any ML model is a sizeable, well annotated dataset.

Our solution–SMT-based race injection. We propose injecting races into existing concurrent software as an approach to scalably create comprehensive, annotated datasets. Specifically, we inject data races into an execution trace of a given program. We choose to inject into the execution trace rather than the source code because injecting races into the program source does not guarantee the race manifesting into every execution trace of the program. This makes it difficult to evaluate race detection algorithms that employ dynamic analysis methods. We refer to the execution trace before injection as the **base trace**.

Adding two consecutive events that are conflicting (*e.g.* a read event immediately followed by a write event to the same variable without any synchronization mechanism) is the simplest way to inject a race into the base trace. More difficult is to inject conflicting events that *could possibly* occur consecutively in a different, random execution of the program. Our goal thus is to generate traces in which such conflicting events appear far apart in them, making them non-trivial to detect. To achieve this, we propose RaceInjector, which injects a trivial data race into any trace, and then uses an SMT-solver to find an alternate, valid reordering of the base trace where the conflicting events appear far apart. This approach is independent of how any race detection algorithm works and the program generating the trace into which the trivial race is injected. Our method importantly guarantees the injection of a race while maintaining the semantics of the base trace. Thus, RaceInjector generates traces with injected races appearing at random, valid locations, mimicking a thread scheduler scheduling a program containing a valid data race. To ensure RaceInjector generates data races that appear arbitrarily far apart in a trace, we propose a method which circumvents practical limitations of SMT-solvers while guaranteeing semantics of the base trace. We

describe our approach in detail in Section 3. In this work, we generate a small dataset and demonstrate it on the research questions that it helps address. We also show how it can be easily extended to a comprehensive dataset. Among the few traces we generate, we find traces with data races that current state-of-the-art race detection methods fail to detect. This demonstrates one immediate utility of RaceInjector. A sample of these counterexamples can be found at https://github.com/ALFA-group/RaceInjector-counterexamples. In Section 4, we discuss other implications of RaceInjector.

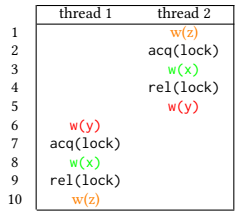


Figure 1. Example of a *potential* data race on lines 1 and 10, an *observed* data race on lines 5 and 6, and a safe access on lines 3 and 8.

2 Background

In this section, we provide a brief background on traces, data races, and the race detection algorithms that we evaluate in this work.

Traces. We assume a sequential consistency memory model [18], where a program trace is a sequence of events on executing a program. An event can be denoted as a tuple <op, thread, loc>, where op is the operation that is performed, thread is the thread which performed the operation, and loc is the file and line which performed the event. An op can be one of the following: `read(x)`, `write(x)`, `lock(L)` (thread has acquired a lock on `L`), `unlock(L)` (thread has released the lock on `L`), `fork(T)` (a thread has forked a new thread `T`), `join(T)` (a thread `T` has joined the current thread). Nondeterminism in the scheduler can cause one program to have many possible traces.

Correct reordering of traces. A trace σ^* is said to be a *correct reordering* of trace σ if it has the following properties:

1. *Thread Ordering:* The order of intra-thread events remains the same in both σ and σ^*.
2. *Read-Write Consistency:* For every read event in σ and σ^*, the most recent write event to the variable that is read remains the same. This is to ensure that control flow will remain the same.
3. *Locking Semantics:* σ^* does not violate the semantics of synchronization events, such as locks and unlocks.

Intuitively, σ^* is a correct reordering of σ if any program that produces σ can also produce σ^*.

Data races. A data race occurs when two threads access the same variable without any synchronization, where at least one of these accesses is a write. Data races can be classified as *observed* or *potential* data races. An *observed* data race is where a data race actively occurs in a trace, where two threads attempt to concurrently access a shared variable where at least one of the accesses is a write. Observed data races are trivial to detect. A *potential* data race is where no data race is observed, but there exists another correct reordering of the trace where an observed data race could occur. Potential data races are much harder to detect. See Figure 1 for an example. Events 5 and 6 in red are an example of an observed race, where two conflicting events occur simultaneously. Events 1 and 10 in orange are an example of a potential race, where they do not occur consecutively in this trace, but could in another correct reordering. Events 3 and 8 are not racy due to synchronization mechanisms.

Data race detection. We evaluate the following algorithms in our work: Happens-Before (Lamport, 1976 [19]), Schedulable Happens-Before (Mathur *et.al.*, 2018 [22]), Weak Causally Precedes (Kini *et.al.*, 2017 [17]), and SyncP (Mathur *et.al.*, 2021 [24]). Happens-Before (HB) creates a partial ordering in a trace between each intra-thread event, as well as between any critical regions, in the order of their appearance in the trace. A partial order on a set S is a relation on S that is reflexive, anti-symmetric, and transitive. Weak Causally-Precedes (WCP) is a weakening of HB, meaning there are fewer edges. This allows WCP to catch more races while maintaining soundness for the first race. WCP only draws edges between critical regions that have conflicting accesses to a shared variable, and draws the edges between the release events and the critical events. Schedulable Happens-Before (SHB) is a strengthening of HB, and discovers fewer races than HB. However, SHB guarantees soundness past the first race. SyncP performs a scan for any data races that do not reverse the order of any critical sections, and is not a partial order. Unlike partial order based techniques, SyncP is unable to detect any races that would require critical sections to be reversed. However, it is guaranteed to catch all races which do not need to reorder the critical sections.

3 Method

In this section, we describe how we inject synthetic data races into program traces. We begin with a motivating example.

Motivating example. The program $\mathcal{P}$ in Figure 2a reads and writes to two variables x and y. One of its possible execution traces is shown in Figure 2b. This is the **base trace** for our injection. Originally, this program does not contain a data race. We note that a thread switch occurs after event 6. We can trivially inject a race directly after the thread switch by adding two write events to the trace (lines 5-6, Figure 2c), resulting in an *observed* data race.

To invoke non-trivial reasoning to detect our injected data race, we propose using an SMT-solver to find a correct reordering of the events in a trace such that (a) the original trace's semantics hold, and (b) the inserted events are *moved*

```
1  class Test {
2        static int x;
3        static int y;
4
5        void inc1() {
6             synchronized(lock){
7                  x++;
8             }
9        }
10
11       void inc2() {
12            synchronized(lock){
13                 y++;
14            }
15       }
16   }
17
18 public static void main(String[] args) {
19   Test test = new Test();
20   fork { test.inc1(); }
21   fork { test.inc2(); }
22 }
```

(a) A simple program $\mathcal{P}$ with two threads and no pre-existing data races

	thread 1	thread 2
1		acq(lock)
2		r(x)
3		w(x)
4		rel(lock)
5	acq(lock)	
6	r(y)	
7	w(y)	
8	rel(lock)	

(b) Original execution trace of $\mathcal{P}$ (Fig 2a)

	thread 1	thread 2
1		acq(lock)
2		r(x)
3		w(x)
4		rel(lock)
5		write(z)
6	write(z)	
7	acq(lock)	
8	r(y)	
9	w(y)	
10	rel(lock)	

(c) Execution trace for Figure 2a, with a trivial race injected.

	thread 1	thread 2
1	write(z)	
2	acq(lock)	
3	r(y)	
4	w(y)	
5	rel(lock)	
6		acq(lock)
7		r(x)
8		w(x)
9		rel(lock)
10		write(z)

(d) The trace after running a solver to move the racy events apart.

Figure 2. A sample program $\mathcal{P}$ (left), and how we can inject a data race in its execution trace (right).

apart by some L events. One such alternate reordering can be seen in Figure 2d. Recall the definition of a data race: two events that access the same variable, at least one of which is a write, that occur in an unsynchronized manner. Our solution to generating these data races has three steps which we describe in detail: **Step 1:** Instrument and execute a program; collect base traces of relevant events. **Step 2:** Add a trivial data race to a base trace, and finally, **Step 3:** Use an SMT-solver to make the added race harder to detect.

Step 1. Trace collection. We start by logging a sequential trace of data accesses and thread synchronizations in a program. See Figure 2b for an example trace. Races are then injected into the collected *base traces* and analyzed. This decoupling of instrumentation and race injection allows for several instrumentation frameworks to be used. We use MCR [10] which instruments using the ASM framework [1]; RoadRunner [8] which also instruments with ASM, and Calfuzzer [15] which instruments using the SOOT compiler framework [31]. SOOT and ASM allow the instrumentation frameworks to modify the bytecode and intercept relevant events as they occur during execution.

Step 2. Adding a trivial race. To modify a base trace to add a trivial data race, we insert two new write events right where there is a context switch between threads. See Figure 2c for an example of modifying the base trace in Figure 2b. The writes are made to a new, dummy variable to ensure the semantics of the original program remains the same. We only inject one race into the base trace at a time before saving it.

Step 3. Using an SMT-solver to move apart the added race. After having injected a trivial race comprising consecutive conflicting events, the goal is to then find an alternate valid interleaving where the race-events are farther apart.

We set up n SMT variables v, where each variable $v_j \in [1, n]$ corresponds to an event that appears in the base trace containing a total of n events. The value of v_j signifies the location index where the event should appear. In trace 2b for example, if v_1 corresponds to event `w(x)`, the assignment for v_1 corresponding to the trace would be 3, the location index `w(x)` appears in the trace. Similarly, if v_2 corresponds to event `w(y)`, the assignment of v_2 corresponding to the trace is 7. For a trace σ, we then formulate a constraint equation in a way that solving the constraints yields a valid assignment made to each v_j which results in an alternate trace σ^*. Our constraints must ensure that the alternate trace is a correct reordering as defined in Section 2 (thread ordering, read-write consistency, locking semantics). These constraints have been commonly defined in race detection to find alternate reorderings [12, 27, 32]. Readers can refer to Said et al. [27] for details. However, we introduce the following constraints in order to inject data races into base traces:

- **Distance between conflicting events.** We supply a hyperparameter `L` which constrains the distance between the inserted racy events.
- **Additional constraints.** We additionally ensure that the indices assigned to each v_j is positive, unique, and lies in the interval $[1, n]$.

We supply a conjunction of these constraints to an SMT-solver which produces an assignment to each v_j. These assignments correspond to a new, valid reordering of each

event appearing in σ, thus resulting in a new trace σ^*. Further, σ^* contains the previously trivially injected race events now at least L events apart, and ensures the same execution semantics as that of σ. We elide details of the symbolic encodings of these constraints for the sake of brevity.

Moving events arbitrarily apart in a trace. The number of constraints in the conjunction described above which generates σ^* is typically prohibitively large for SMT-solvers to solve. Our insight to circumvent this practical problem is to incrementally move the introduced conflicting events farther apart. We start with reordering the conflicting events (which initially appear consecutively when injected) and the events surrounding it in a window of fixed size. For the events in this window, we generate the constraints described above and run RaceInjector. We choose a window size in a way that the number of constraints does not overwhelm the solver. Once RaceInjector generates a reordering for the events in the window, we slide the window over by a fixed length and run RaceInjector again on the events that appear in the shifted window. We ensure the shifted window contains at least one of the two conflicting events we introduce, which will have been reordered from their initial, consecutive indices. Running RaceInjector iteratively over smaller, fixed-length windows k times is computationally much more efficient than running the solver on a large number of events just once—the number of constraints tend to grow superlinearly with the number of events needed to reason about. We use a window size of 100 in our implementation. We elide a proof of correctness of our approach for the sake of brevity.

4 Results & Discussion

We demonstrate RaceInjector by using it with a suite of program benchmarks used in prior works to generate a sample of base traces containing data races (Section 4.1). Among the generated traces, we also find counterexamples which state-of-the-art race detection algorithms fail to detect (Section 4.2).

4.1 Generated Dataset: Quantitative Description

We employ RaceInjector to generate only a small, demonstrative dataset comprising ~1000 total traces in this work. This is nonetheless sufficient to show the ease with which RaceInjector can be extended to generate a comprehensive dataset. We ran our experiments without any parallelization using Java version 11.0.18, on a CPU running Ubuntu 18.04 with 96 GB RAM.

Base traces. We run each of the five program benchmarks listed in Table 1 once on the testcases from `Calfuzzer` [15]. This instruments and generates one execution trace each for each of the five programs. Table 1, columns 2, 3 and 4 document statistics of each program's base trace. Each program has a different number of threads (column 4). Consequently, each trace presents a different number of points of injection (column 2) to introduce a trivial race—these are points at which thread context-switches occur. We run RaceInjector on each program's trace (except for Jigsaw) for one hour, with a goal of injecting races into as many entry points as possible within the allotted one hour. Since Jigsaw is a significantly larger program than the rest, as seen by the length of an average trace generated (column 3), we run RaceInjector for ~10 hours instead on the Jigsaw trace.

RaceInjector-generated traces. Running RaceInjector results in a total of ~1000 traces (sum of column 5). Columns 5, 6 and 7 in Table 1 document the statistics of the generated traces. Note again, these are all guaranteed to contain data races. The set of traces generated by RaceInjector for any one program will all have the same length (column 3) because each trace is just one possible valid reordering of the original. We find the number of traces (column 5) generated in one hour of running RaceInjector is roughly the same across the different program benchmarks (ignoring Jigsaw). In columns 6 and 7, we report the average distance and the maximum distance between the injected conflicting events in the traces generated by RaceInjector, which is measured by counting the number of events between them. We observe the average distances (column 6) to be significantly greater than zero, suggesting that the injected races, which are initially placed consecutively, end up significantly apart in the generated traces. From the standard deviations (subscripts in column 6), we see very high variance in the distance between injected races, suggesting the heterogeneity in the potential data races introduced by RaceInjector. In `ArrayList` and `TreeSet`, the maximum distance between the injected race events (column 7) nearly span the length of their base traces (column 3). This demonstrates the flexibility offered by RaceInjector to generate any number of traces with guaranteed races that are varied in the locations they appear in. This is a desirable feature for a high-quality annotated dataset of such concurrent programs. It is important to note that distance is not the sole determinant of whether a data race is difficult to detect. It is possible for a program to have many events, but with no synchronization mechanisms. In this case, races that are far apart would still be trivial to detect. However, detecting races that are far apart has been shown to be difficult for previous algorithms [24].

Discussion: Scaling the dataset. To assemble a comprehensive dataset, we recommend the following procedure: first, execute a program multiple times to obtain different base traces. Second, run RaceInjector on every possible point of injection in each base trace without constraining the time taken to complete this process. Both the steps are easy to parallelize. We plan to extend the race detection algorithms evaluated to include more recent tools such as RPT [30], static race detection methods like Infer [6], and SMT-based methods [12].

Table 1. Overview of RaceInjector results on a benchmark of programs. Column 1 lists the different program benchmarks in which RaceInjector injects races. Columns 2,3,4 describe the base traces. The remaining columns describe the traces generated by RaceInjector. *Inj. pts.* refers to the number of injection points available in the base trace; *Thrd* is the number of program threads.

	Base traces			RaceInjector-generated traces		
Program	#Inj. pts	Length	#Thrd	#Gen. traces	Avg race dist.	Max race dist.
ArrayList	207	677	27	207	$128_{\pm 111}$	558
TreeSet	130	756	22	130	$122_{\pm 115}$	526
LinkedList	1767	14937	451	160	$112_{\pm 124}$	851
Stack	2036	11372	451	100	$87_{\pm 74}$	458
Jigsaw	3394	97110	78	467	$693_{\pm 777}$	7396

Discussion: True-negative samples. In proposing RaceInjector, we only consider the problem of generating true-positive data races. In our larger goal to assemble a comprehensive dataset large enough to train machine learning models, we would also need a method to assemble examples of true-negative cases in our dataset. As most pairs of conflicting accesses in software are not races, we can randomly sample such accesses, verify them using simple algorithms like HB, and label them as true-negatives. This does not guarantee that any given pair of conflicting accesses are not racy, thus we are unable to evaluate the false positive rates of heuristic based algorithms without additional manual analysis. However, machine learning algorithms are tolerant to a small number of mislabeled samples.

4.2 Counterexamples to SOTA Algorithms

Table 2 shows that RaceInjector can easily produce counterexamples to state-of-the-art (SOTA) detection algorithms. This is important because it reveals for the first time their sensitivity as well as guides future work to clearly define new rules and algorithms. We can potentially study the contribution of the different rules present in the heuristics of different algorithms in the decisions made by the algorithms. Thus, access to a comprehensive set of counterexamples can potentially empirically justify the different rules implemented by these algorithms, and can also point to equivalences between some of these rules. While we do not directly study the counterexamples, this is a compelling direction for future work.

We now analyze the counterexamples generated by RaceInjector that the different algorithms fail to detect. They are available at: https://github.com/ALFA-group/RaceInjector-counterexamples. The analysis that follows should be interpreted with caution because the number of counterexamples is relatively small (fewer than 100), which does not support statistical comparison tests. We will evaluate these claims rigorously on a larger dataset in future work. This analysis is instead indicative of the questions that can be studied.

Table 2. Counterexamples generated by RaceInjector. A ✔ signifies there exists at least one trace among the RaceInjector-generated traces which is not detected by the corresponding algorithm. *# Missed* reports the number of traces the algorithm misses to detect (percentage mentioned within parenthesis).

Algorithm	ArrayList	TreeSet	Jigsaw	Stack	LinkedList	# Missed
HB (1979) [19]	✔	✔	✔	✘	✘	60 (5.6%)
SHB (2018) [22]	✔	✔	✔	✘	✘	64 (6%)
WCP (2017) [17]	✘	✔	✘	✘	✘	21 (2%)
SyncP (2020) [24]	✔	✔	✔	✘	✘	22 (2%)

SHB vs. HB. SHB guarantees soundness after the first detected data race it detects in exchange for detecting fewer races overall compared to HB. We should then expect SHB to detect fewer races on average, and conversely miss detecting more races. This is what we observe: SHB fails to detect 64 bugs RaceInjector generates (column 7, Table 2), 4 more than HB. That said, the total percentage of bugs that the algorithms fail to detect are roughly the same (~6%). We will, in the future, also compare whether they fail on the same set of counterexamples.

SyncP vs. WCP. Despite SyncP following and improving upon WCP, we do not see a notable increase in its performance. Both fail to detect 2% of the generated races. On the other hand, both algorithms improve upon HB, so it is expected to see a improvement of ~4% in the races they fail to detect when compared to HB.

`LinkedList` *and* `Stack`. We observe that none of the injected data races in `LinkedList` and `Stack` fail any algorithms (columns 5, 6, Table 2). Besides the low number of samples generated, a possible reason could be the large number of unsynchronized threads. These two programs have the largest number of threads relative to the length of their traces (Table 1). We suspect these threads mostly involve unsynchronized accesses, making the injected races relatively easy to detect as well. If in the future our hypothesis that the threads mostly involve unsynchronized accesses holds, we will filter out such injection points to reduce the number of trivially detectable races in our dataset.

Table 2 indicates that RaceInjector is able to generate data races which no SOTA method detects. This implies RaceInjector finds locations in a program trace which are complex to reason about. To finish, RaceInjector makes the widespread adoption of classification accuracy-related metrics (true-positive, false-positive, true-negative, false-negative) now possible when evaluating and comparing race detection algorithms.

5 Related Work

Prior work closest to RaceInjector has mostly compiled known bugs that have been found over the years. Because these bugs have already been found, it is difficult to evaluate the capability of new approaches to detect new bugs. Additionally, these datasets are far too small to train a machine learning model, the largest being 985 races in Jbench [9]. JaConTeBe [20] is a benchmark of Java concurrency bugs, which scrapes past papers and aggregates a list of 47 distinct bugs along with their causes. GoBench [33] is a dataset of 103 bugs in Go, scraped from Github. RADBench [13] is a dataset composed of snapshots of open-source software projects with 10 total known bugs. Jbench [9] is a dataset of Java data races, aggregated from artifacts of existing race detection tools, and contains 985 unique data races. Jbench contains 6 real-world applications, and 42 custom testcases that were written during development of previous race detection tools. Typically, all these benchmarks are curated by either expensive manual analysis, or have been assembled using existing tools, which greatly limits their usefulness in evaluating and improving extant race detection algorithms while RaceInjector is fully automated. Additionally, since many of the samples have been curated using existing tools, a machine learning model trained on these samples will be unlikely to outperform the original tools used to find them.

There are also existing works on injecting synthetic bugs into programs to train machine learning models [25], as well as evaluating synthetic bugs [4]. In the future, a more robust evaluation of RaceInjector-generated bugs will be necessary to ensure they are representative of real-world race conditions.

Acknowledgments

This work was partially supported by a grant from the MIT-IBM Watson AI lab.

References

[1] [n. d.]. ASM bytecode analysis framework. https://asm.ow2.io/

[2] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[3] Benjamin Bowman, Craig Laprade, Yuede Ji, and H Howie Huang. 2020. Detecting Lateral Movement in Enterprise Computer Networks with Unsupervised Graph AI.. In *RAID*. 257–268.

[4] Joshua Bundt, Andrew Fasano, Brendan Dolan-Gavitt, William Robertson, and Tim Leek. 2021. Evaluating Synthetic Bugs. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security* (Virtual Event, Hong Kong) *(ASIA CCS '21)*. Association for Computing Machinery, New York, NY, USA, 716–730. https://doi.org/10.1145/3433210.3453096

[5] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael F P O'Boyle, and Hugh Leather. 2021. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 2244–2253. https://proceedings.mlr.press/v139/cummins21a.html

[6] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70.

[7] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

[8] Cormac Flanagan and Stephen N. Freund. 2010. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Toronto, Ontario, Canada) *(PASTE '10)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/1806672.1806674

[9] Jian Gao, Xin Yang, Yu Jiang, Han Liu, Weiliang Ying, and Xian Zhang. 2018. Jbench: A Dataset of Data Races for Concurrency Testing. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 6–9. https://doi.org/10.1145/3196398.3196451

[10] Jeff Huang. 2015. Stateless Model Checking Concurrent Programs with Maximal Causality Reduction. *SIGPLAN Not.* 50, 6 (jun 2015), 165–174. https://doi.org/10.1145/2813885.2737975

[11] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal sound predictive race detection with control flow abstraction. In *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*. 337–348.

[12] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. *SIGPLAN Not.* 49, 6 (jun 2014), 337–348. https://doi.org/10.1145/2666356.2594315

[13] Nicholas Jalbert, Cristiano Pereira, Gilles Pokam, and Koushik Sen. 2011. RADBench: A Concurrency Bug Benchmark Suite. In *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar 11)*. USENIX Association, Berkeley, CA. https://www.usenix.org/conference/hotpar-11/radbench-concurrency-bug-benchmark-suite

[14] Minseok Jeon, Myungho Lee, and Hakjoo Oh. 2020. Learning graph-based heuristics for pointer analysis without handcrafting application-specific features. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[15] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. 2009. CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs. In *Computer Aided Verification*, Ahmed Bouajjani and Oded Maler (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 675–681.

[16] Christian Gram Kalhauge and Jens Palsberg. 2018. Sound Deadlock Prediction. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 146 (oct 2018), 29 pages. https://doi.org/10.1145/3276516

[17] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic Race Prediction in Linear Time. *SIGPLAN Not.* 52, 6 (jun 2017), 157–170. https://doi.org/10.1145/3140587.3062374

[18] Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* C-28, 9 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

[19] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (jul 1978), 558–565. https://doi.org/10.1145/359545.359563

[20] Ziyi Lin, Darko Marinov, Hao Zhong, Yuting Chen, and Jianjun Zhao. 2015. JaConTeBe: A Benchmark Suite of Real-World Java Concurrency Bugs (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 178–189. https://doi.org/10.1109/ASE.2015.87

[21] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 664–676.

[22] Umang Mathur, Dileep Kini, and Mahesh Viswanathan. 2018. What happens-after the first race? enhancing the predictive power of happens-before based dynamic race detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29.

[23] Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2020. The Complexity of Dynamic Data Race Prediction. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science* (Saarbrücken, Germany) *(LICS '20)*. Association for Computing Machinery, New York, NY, USA, 713–727. https://doi.org/10.1145/3373718.3394783

[24] Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2021. Optimal Prediction of Synchronization-Preserving Races. *Proc. ACM Program. Lang.* 5, POPL, Article 36 (jan 2021), 29 pages. https://doi.org/10.1145/3434317

[25] Jibesh Patra and Michael Pradel. 2021. Semantic Bug Seeding: A Learning-Based Approach for Creating Realistic Bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 906–918. https://doi.org/10.1145/3468264.3468623

[26] Jake Roemer, Kaan Genç, and Michael D Bond. 2020. SmartTrack: efficient predictive race detection. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 747–762.

[27] Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. 2011. Generating Data Race Witnesses by an SMT-Based Analysis. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 313–327.

[28] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 391–411.

[29] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. *Sound Predictive Race Detection in Polynomial Time*. Association for Computing Machinery, New York, NY, USA, 387–400. https://doi.org/10.1145/2103656.2103702

[30] Mosaad Al Thokair, Minjian Zhang, Umang Mathur, and Mahesh Viswanathan. 2023. Dynamic Race Detection with O(1) Samples. *Proc. ACM Program. Lang.* 7, POPL, Article 45 (jan 2023), 30 pages. https://doi.org/10.1145/3571238

[31] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (Mississauga, Ontario, Canada) *(CASCON '99)*. IBM Press, 13.

[32] Chao Wang, Sudipta Kundu, Malay Ganai, and Aarti Gupta. 2009. Symbolic predictive analysis for concurrent programs. In *International Symposium on Formal Methods*. Springer, 256–272.

[33] Ting Yuan, Guangwei Li, Jie Lu, Chen Liu, Lian Li, and Jingling Xue. 2021. GoBench: A Benchmark Suite of Real-World Go Concurrency Bugs. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 187–199. https://doi.org/10.1109/CGO51591.2021.9370317

Received 2023-03-10; accepted 2023-04-21

Author Index