BinTrimmer: Towards Static Binary Debloating Through Abstract Interpretation

Nilo Redini[1], Ruoyu Wang[2], Aravind Machiry[1], Yan Shoshitaishvili[2], Giovanni Vigna[1], and Christopher Kruegel[1]

[1] UC Santa Barbara
{nredini,machiry,vigna,chris}@cs.ucsb.edu
[2] Arizona State University
{fishw,yans}@asu.edu

Abstract. The increasing complexity of modern programs motivates software engineers to often rely on the support of third-party libraries. Although this practice allows application developers to achieve a compelling time-to-market, it often makes the final product *bloated* with conspicuous chunks of *unused* code. Other than making a program unnecessarily large, this dormant code could be leveraged by willful attackers to harm users. As a consequence, several techniques have been recently proposed to perform *program debloating* and remove (or secure) dead code from applications. However, state-of-the-art approaches are either based on unsound strategies, thus producing unreliable results, or pose too strict assumptions on the program itself.

In this work, we propose a novel abstract domain, called *Signedness-Agnostic Strided Interval*, which we use as the cornerstone to design a novel and sound static technique, based on abstract interpretation, to *reliably* perform program debloating. Throughout the paper, we detail the specifics of our approach and show its effectiveness and usefulness by implementing it in a tool, called BinTrimmer, to perform static program debloating on binaries.

Our evaluation shows that BinTrimmer can remove up to 65.6% of a library's code and that our domain is, on average, 98% more precise than the related work.

1 Introduction

Computer applications and services are continuously getting more sophisticated, and, as a result, their software is becoming more complex. Besides, the attempt to reduce the time-to-market is putting software engineers under an enormous amount of pressure. As a result, more and more software developers choose to rely on the help of ready-to-use *third-party libraries* to implement complex software functionality. Since third-party libraries are meant to be used by a wide variety of applications, a specific program relying on them does not commonly use *all* of the library functionality. That is, there exists code in the third-party library that is superfluous for the final application. Other than merely making a program unnecessarily big, this dead code is potentially dangerous as it increases the *surface* (i.e., the amount of code) an attacker has to harm the users. In fact, if the main application has a vulnerability that can be used to redirect the program execution to the dead code, this code could be leveraged by an attacker to gain greater capabilities. The process of decreasing the attack surface of a program by inhibiting the execution of its dead code is called *program debloating.*

In the literature, several approaches have been recently proposed to identify and remove [15] (or secure [46]) dead code from programs. Unfortunately, other than posing strong assumptions about the availability of the programs' test cases [25,33], source code [29] and run time support [5], these approaches are hardly employable in practice as they often rely on unsound strategies, and, therefore, unable to guarantee the correct functioning of the debloated program.

Theoretically, perfect program debloating is achieved by identifying and removing *all and only* those portions of code that are unreachable by any execution of a program. Using the definition of *soundness* and *completeness* as defined by Xu et al. in [44], we can reduce this problem to creating the *ideal* (i.e., *complete* and *sound*) Control Flow Graph (CFG) of a given application, and removing all the code not referenced by it. Though the generation of the ideal CFG is proven to be an undecidable problem [23,30], the necessary condition to remove any code from a program while guaranteeing its correctness is for the CFG to be complete. Of course, an increasingly precise CFG (i.e., containing a small number of spurious control-flow transfers) would lead to the removal of more significant portions of dead code. Unfortunately, precisely determining the control-flow transfers of an arbitrary program is challenging, as the program might contain code pointers whose targets are resolved at runtime (*indirect control-flow transfers*). This problem could be solved by computing the exact set of *values* that the program pointers can assume during any execution of the program itself. Unfortunately, this is a hard problem [23,31]. In literature, several techniques [1,12,18,26] have been proposed to approximate the set of values assumed by program variables through their range. However, either they are applicable when the *signedness* of a variable (i.e., `signed` or `unsigned`) is known, which is usually not the case in binary programs, or their results are too imprecise for practical uses.

In this work, we take a step further and propose a novel abstract domain, which we call the *Signedness-Agnostic Strided Interval* (SASI) domain, specifically designed to achieve sound program debloating on binaries. Then, we propose and detail a novel and sound approach that leverages our domain to *safely* perform *static* program debloating on binary files. The advantage of our approach is threefold: First, it can be used on binaries for different architectures. Second, it does not make any assumptions on the availability of test cases or source code, and, therefore, it can be applied to every program. Third, and more importantly, our approach is sound, which means that the correct execution of the debloated program can be mathematically guaranteed. We demonstrate the effectiveness of our approach by implementing it in a tool, called BinTrimmer. To the best of our knowledge, this is the first test-case-agnostic, static debloating technique that works directly on binaries. Furthermore, we show that our new abstract domain, SASI, improves the precision (on average by 98%) of value ranges of all the variables in the program, compared to the related work. We have implemented (and open-sourced) our abstract domain atop two analysis frameworks: LLVM, for source code [3], and `angr`, for binary code [4].

In summary, our contributions are the following:

- We propose the first sound, test-case agnostic program debloating approach for binaries.
- We design and formalize a novel signedness-agnostic abstract domain, which outclasses the related work in terms of both soundness and precision, and implement it in two different frameworks: LLVM (for source code analysis) and `angr` (for binary analysis).
- We implemented our approach in a prototype, called BinTrimmer, that using iterative value-flow refinement, recovers a complete and precise CFG from a binary, identifies unreachable code, and removes it.
- We perform a preliminary evaluation of BinTrimmer on real-world applications and show that our approach is effective at program debloating.
- We extensively evaluate our abstract domain, SASI, against domains proposed in related work on both source code and binary files.

2 Background and Motivation

Value range analysis [35] is a particular type of data-flow analysis that tracks the range of values that a numeric entity (e.g., a program variable) might assume at any point of a program's execution. These analyses are built on top of abstract domains [9,8,13], and can be utilized to guide the recovery of a program's CFG by: (i) helping to determine control dependencies between programs statements, and (ii) resolving the targets of indirect control-flow transfers.

```
void main() {
  uint8_t opt;
  void (*f_ptr)(void) = [foo, bar, baz]; // foo, bar, and baz are
                                        // defined in another module
  scanf("%"SCNu8, &opt);
  opt = (opt * 2) + 1;
  // ...
  if (opt == 0) {
    f_ptr[0](); // call to foo
  } else if(op == 100){
    f_ptr[1](); // call to bar
  } else if (opt > 127) {
    f_ptr[2](); // cal to baz
  }
}
```

Source Code 1.1: Precisely determining variable values is crucial to recover the ideal CFG.

Consider for instance Code 1.1. A sound and precise value range analysis would determine that: (i) The variable `opt` can only assume odd values, and (ii)

[3] https://github.com/ucsb-seclab/sasi

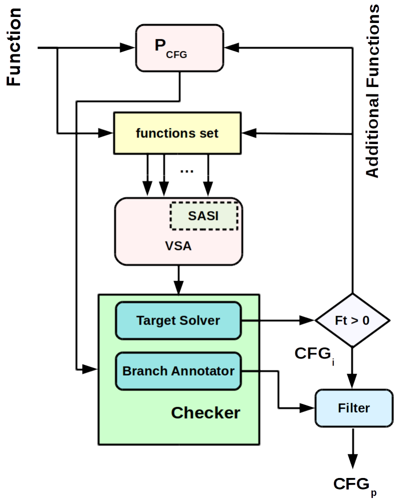
[4] https://github.com/angr/claripy/blob/master/claripy

the function pointer *f_ptr* can point to the functions `foo`, `bar`, and `baz`. A CFG recovery algorithm employing this range analysis would leverage these two pieces of information to retrieve a complete and sound CFG. Precisely, the algorithm would determine that the `if` conditions at Line 8 and Line 10 are never satisfied, and, therefore, that the functions `foo` and `bar` are dead code and they should not appear in the program's CFG.

To recover a complete and (possibly) sound CFG, the CFG recovery algorithm should rely on a sound and precise range analysis. To produce sound results, range analyses must be able to reason about the signedness of program variables. Considering the example in Code 1.1, if a given range analysis a_s assumes incorrectly that the variable `opt` is signed, it would determine that `opt` cannot assume values higher than 127, and, therefore, the `if` condition at Line 12 would be considered unsatisfied under any execution of the program. While the source code of programs written with strong-typed languages (e.g., C/C++) explicitly state a variable signedness, determining such information in binaries is a hard problem [4,24]. In these cases, the solution is to consider each variable as *both* signed and unsigned, that is, to make the domain of each variable in a program *signedness-agnostic*. The first step in this direction has been taken by Navas et al. [27], who proposed an abstract domain called *Wrapped Intervals* (WI), which represents both signed and unsigned numeric values. Albeit sound, Wrapped Intervals produce too imprecise results to be applicable in practice. In fact, in this domain, a variable can assume *any* of the values within a range, whereas, in practice, only some of the values might be assumed during any execution of the program. This imprecision might impact the soundness of a CFG. Consider Code 1.1, and assume that the range analysis employed by the CFG recovery algorithm determines that the variable `opt` can assume every value between 1 and 255. In this case, the CFG recovery algorithm would mistakenly establish that the `if` condition at Line 10 can be satisfied, and, therefore, that the function `bar` should be included in the program's CFG.

In this work, we restore this loss of precision, while maintaining signedness agnosticism, by designing a domain based on the fundamental concepts of Wrapped Intervals, but supporting a *stride*. We call this domain *Signedness-Agnostic Strided Interval*. The use of a stride allows us to precisely determine the values that a program variable can assume (e.g., odd values for `opt` in Code 1.1), thus improving the precision of a program CFG.

Our domain is particularly suited for binary analysis. In fact, there are several high-level code constructs (e.g., `switch-case` statements) that are translated in binary code in a way (e.g., through jump tables) that Wrapped Intervals would not handle well. In these cases, the use of a stride would significantly improve the precision of the overall analysis (e.g., by precisely enumerating the destinations of a jump table).



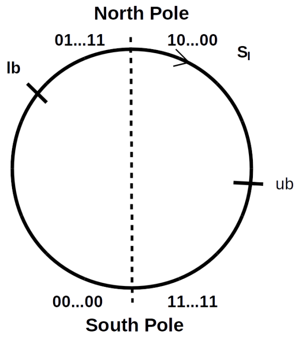


Fig. 1: Iterative CFG Refinement Algorithm. **Fig. 2:** Signed-Agnostic Strided Interval (SASI).

3 Overview

Our approach to soundly perform code debloating of a program P is based on the recovery of a complete and precise CFG for P. Given a program P to debloat, if the CFG G for P is complete, every basic block not present in G can be safely removed from P without hindering its correctness. However, the more G is precise, the more basic blocks can be safely removed from P. In fact, if G is also sound *all* the useless basic blocks would be removed from P. To achieve this goal, we designed a new technique called *Iterative CFG Refinement*.

3.1 Iterative CFG Refinement

Given a function f (e.g., the address of a program's entry point), the Iterative CFG Refinement procedure iteratively builds f's CFG and leverages a sound algorithm based on value-range analysis to refine it.

The Iterative CFG Refinement algorithm relies on the availability of a procedure P_{CFG} to recover the CFG of the function f. We assume that P_{CFG} can recover all the basic blocks and code boundaries within f. We do not make any further assumptions about the precision of P_{CFG}. For example, P_{CFG} could be simply defined as a procedure that creates edges among all the possible basic blocks of f. The iterative CFG refinement algorithm is depicted in Figure 1, and can be divided into three main components, which we explain in the remaining of this section.

CFG and VSA. First, we use P_{CFG} to compute f's CFG, and add f to a *function set* (initially empty). Then, we perform a *Value-Set Analysis* [1] (or *VSA*) on each function in the function set. The VSA is a static analysis based on abstract interpretation [8] that determines a conservative approximation of the set of numeric values and addresses that variables assume at each program point within a function f. The VSA utilizes our abstract domain SASI (detailed in Section 4) to analyze f and retrieve precise information about the binary variables (i.e., registers and memory locations).

Checker. The *Checker* module utilizes the VSA results to augment and refine the CFG through two different sub-modules: the *Branch Annotator*, and the *Target Solver*.

The *Branch Annotator* retrieves each CFG's conditional edge e_c (i.e., guarded by an *if-then-else* condition), and analyzes the logical expression of the condition that determines whether e_c would be taken or not at runtime. To this end, it relies on the abstract operations defined on SASI (shown in Appendix A) to evaluate the theoretical satisfiability of the expression. If no solution exists, the Branch Annotator annotates e_c and marks it for removal.

On the other hand, the *Target Solver* considers f's basic blocks and collects those having indirect control-flow transfers (e.g., due to an indirect call). It then uses the VSA information to gather the set of function targets to which each indirect flow transfer can resolve, and add them to a set Ft. These functions are used to recover a new augmented CFG and to bootstrap a new round of VSA.

When a fixed-point is reached, that is when no new flow transitions are discovered (i.e., $Ft = 0$) and no new edge is annotated, the current CFG (i.e., CFG_i) is passed to the *Filter* module.

Filter. The Filter module scans every edge in CFG_i and removes each annotated edge. Then, for each basic block b in CFG_i, it checks whether it exists an inbound edge for b. If not, it retrieves all the nodes dominated (as defined in graph theory [43]) by b and removes them from CFG_i. Finally, it returns the filtered CFG_p.

3.2 Program Debloating

From a security point of view, the problem of program debloating is formulated as *decreasing the attack surface* of a program by removing its dead code. This goal can be achieved with two different techniques: (i) *deleting* the dead code from the binary, (ii) *rewriting* the dead code with useless instructions (e.g., `hlt`).

Though both approaches effectively remove the potentially dangerous dead code from a program, the former presents more challenges. In fact, if the code of a binary is modified, potentially all of its code and data pointers must be updated to reflect the new program layout. In literature, two main approaches are proposed to achieve this goal: *Binary Instrumentation* and *Binary Rewriting*. In the former approach, a binary file is usually augmented with pieces of trampoline code that fix the program pointers at runtime [3,14,28,46]. In the latter approach, Binary Rewriting techniques [40,41] attempt to achieve perfect disassembling (i.e., by solving code and data pointers), thus being able to recompile a program. Unfortunately, any of the techniques mentioned above present several limitations and trade-offs (e.g., ignoring computed code pointers) that hinder their soundness.

For this reason, to preserve the soundness of our approach, we decided to eliminate the dead code of a program by rewriting it. This approach, though not decreasing the size of a program itself, presents mainly two advantages: (i) the program does not need any external support to be executed (e.g., a modified dynamic loader to perform runtime address resolution) and (ii) soundness is preserved. Note, however, that our approach can be easily extended to use one of the state-of-the-art solutions of binary rewriting, such as Ramblr [40], to effectively delete the dead code.

4 Signedness-Agnostic Strided Intervals

In this section, we present a novel approach to the abstract modeling of numeric entities with a fixed width. We define a new *abstract domain* named *Signedness-Agnostic Strided Interval* to represent the set of values that a numeric entity (of a given bit-width) can possibly assume.

4.1 Definition

A Signed-Agnostic Strided Interval (abbreviated *SASI*) is indicated as $r = s_r[lb, ub]w$, where lb and ub are bit-vectors of w bits (called *lower bound* and *upper bound* respectively), whereas s_r (called *stride*), is a non-negative integer.

A SASI r represents the set of values: $\{lb,\ lb{+_w}s_r,\ lb{+_w}2{*}s_r,\ \ldots,\ ub\}$, where $+_w$ represents modular addition of bit-width w (i.e $x +_w y = (x + y) \bmod 2^w$). Formally,

$$r = \{(lb + k * s_r) \bmod 2^w \leq ub,\ \ k \in \mathbb{N}\} \tag{1}$$

For example, `2[1010, 0010]4` represents the set of values {`1010, 1100, 1110, 0000, 0010`}. Note that, the SASI $0[lb, lb]w$ represents the singleton lb.

A SASI variable can be graphically represented through a *number circle*, as depicted in Figure 2. The set of numerical values represented by a SASI are determined by traversing the number circle clockwise starting from the lower bound lb up to the upper bound ub with increments of the stride value s_r. SASI can represent *unsigned* and *signed* variables alike.

For example, consider the SASI $r = 1[0100, 1010]4$ representing a variable x (i.e., $x \in r$). r represents the values $4 \leq x \leq 10$ if x is interpreted as an unsigned variable, or the values $(4 \leq x \leq 7) \vee (-8 \leq x \leq -6)$ if interpreted as signed. In the case of signed values, the *South Pole* and the *North Pole* divide the positive and negative numbers: Positive numbers begin from the left of South Pole, proceeding clockwise up to the left of North Pole. Similarly, negative values begin from the right of the North Pole, proceeding clockwise down to the right of South Pole. Note that, operations on SASI (Appendix A) do not assume the signedness of variables, thus providing sound results for both signed and unsigned interpretations.

Throughout this work we use the following notation: B_w and W_w indicate the set of all the possible bit-vectors representable on w bits, and the set of all the possible SASIs representable on the same number of bits, respectively. A modular operation on w bits is indicated as op_w (e.g., $+_w$), where $x\ op_w\ y = (x\ op\ y)$ mod 2^w. We use the sequence representation b^k to express a k-long sequence of the bit b ($b \in \{0, 1\}$), and the symbol $||$ to indicate sequence concatenation. Furthermore, the symbol $\leq$ represents the lexicographic ordering in B_w, whereas $\leq_x$ represents the relative ordering, with respect to the value x, on the number circle (Figure 2). That is to say:

$$a \leq_x b \;\; iff \;\; (a -_w x) \leq (b -_w x) \tag{2}$$

Informally, starting from x and proceeding clockwise on a number circle, a is encountered before b.

Using the above notations, we now define several functions as needed for any static analysis based on abstract interpretation.

Definition 1. ***Concretization Function.*** *Given a SASI $r = s_r[lb, ub]w$, the concretization function $\gamma : W_w \rightarrow P(B_w)$ is defined as follows:*

$$\begin{aligned} &\gamma(\bot) = \emptyset \\ &\gamma(r) = \{lb,\ lb +_w s_r,\ lb +_w 2 * s_r,\ ...,\ ub\} \\ &\gamma(\top) = B_w \end{aligned} \tag{3}$$

Where $P(B_w)$ is the power set of B_w, $\bot$ denotes an empty SASI (i.e., $0[,]w$) and $\top$ denotes the full SASI (i.e., $1[0^w, 1^w]w$).

Definition 2. ***Abstraction Function.*** *Given a set of values $V = \{v_1, v_2, ..., v_n\}$, the abstraction function $\alpha : P(B_w) \rightarrow W_w$ is defined as follows:*

$$\begin{aligned} &\alpha(\emptyset) = \bot \\ &\alpha(V) = s_r[a_1, a_n]w, (a_j)_{j=1}^n = sort(v_1, v_2, ..., v_n) \\ &\alpha(B_w) = \top \end{aligned} \tag{4}$$

where $s_r = gcd(d_1, d_2, ..., d_{n-1})$ and $d_j = a_{j+1} -_w a_j,\ for\ 1 \leq j \leq (n-1)$. gcd is the greatest common divisor function, and $sort$ is a function sorting values in ascending order.

Intuitively, given a set of bit-vectors, the abstraction function sorts its elements in ascending order, thus creating the sequence $(a_j)_{j=1}^n$. Then, it considers the first and last elements as the lower and upper bounds respectively, and, starting from the lower bound, it selects the greatest stride s_r that includes all the elements in $(a_j)_{j=1}^n$.

Definition 3. ***Membership Function.*** *Given a bit-vector v and a SASI $r = s_r[lb, ub]w$, the membership function $\in$ is defined as follows:*

$$v \in r = \begin{cases} true & if\ r = \top \\ false & if\ r = \bot \\ v \leq_{lb} ub \wedge (v -_w lb) mod\ s_r = 0 & if\ r = s_r[lb, ub]w \end{cases} \tag{5}$$

Definition 4. ***Cardinality Function.*** *Given a SASI* $r = s_r[lb, ub]w$ *the cardinality function* $\#$ *is defined as:*

$$\begin{aligned}
\#(\bot) &= 0\\
\#(\top) &= 2^w\\
\#(r) &= \left\lfloor \frac{ub - lb + 1}{s_r} \right\rfloor
\end{aligned}$$

Definition 5. ***Ordering Operator.*** *Given two SASIs* $r = s_r[a, b]w$ *and* $t = s_t[c, d]w$*, the ordering operator* $\sqsubseteq$ *is defined as follows:*

$$r \sqsubseteq t = \begin{cases}
False & if\ r = \top \wedge t \neq \top\\
True & if\ r = \bot \vee\ t = \top\ \vee\\
 & \quad ((a = c) \wedge (b = d)\ \wedge\\
 & \quad\ (s_r\ mod\ s_t = 0))\\
a \in t \wedge b \in t\ \ \wedge (c \notin r \vee d \notin r) & \\
\quad \wedge (a - c)\ mod\ s_t = 0 & otherwise\\
\quad \wedge\ s_r\ mod\ s_t = 0 &
\end{cases} \qquad (6)$$

In other words, one SASI is considered to be *included* in another if every value in the former is contained in the latter, that is $\gamma(r) \subseteq \gamma(t)$.

Note that, while $(\sqsubseteq, W_w)$ forms a partially ordered set (with least element $\bot$ and greatest element $\top$), it does not form a lattice as the ordering does not always provide a unique *least upper bound* (or *join*) and *greatest lower bound* (or *meet*). For example, consider the two SASIs $2[0010, 0100]4$ and $2[1000, 1110]4$. Two minimum upper-bounds (i.e., having the same cardinality) for these SASIs are $2[0010, 1110]4$ and $2[1000, 0100]4$. However, they are incomparable, thus violating the unique least upper bound requirement. Since a join and meet are not available, we must define a deterministic pseudo-join and a pseudo-meet. For spaces reasons, we present in this paper only the pseudo-join operator.

Definition 6. ***Pseudo-Join Operator.*** *Given two SASI* $r = s_r[a, b]w$ *and* $t = s_t[c, d]w$*, the pseudo-join operator* $\tilde{\sqcup}$ *is defined as follows:*

$$r \mathbin{\tilde{\sqcup}} t = \begin{cases}
t & if\ r \sqsubseteq t\\
r & if\ t \sqsubseteq r\\
\top & if\ a \in t \wedge b \in t \wedge c \in r \wedge d \in r\\
s_{ad}[a, d]w & if\ c \in r \wedge b \in t \wedge a \notin t \wedge d \notin r\\
s_{cb}[c, b]w & if\ a \in t \wedge d \in r \wedge c \notin r \wedge b \notin t\\
s_{ad}[a, d]w & if\ a \notin t \wedge d \notin r \wedge c \notin r \wedge b \notin t\ \wedge\\
 & \qquad \#(s_{ad}[a, d]w) \leq \#(s_{cb}[c, b]w)\\
s_{cb}[c, b]w & otherwise
\end{cases} \qquad (7)$$

Where $s_{xy} = gcd(s_r, s_t, y -_w x)$, with $xy \in \{(a, d), (c, b)\}$ and gcd is the great common divisor function.

The pseudo-join operator we defined assures that the SASI with the lowest cardinality, and, thus, most precise, is always picked. However, it is not associative, that is $((r \mathbin{\tilde{\sqcup}} t) \mathbin{\tilde{\sqcup}} z) \neq (r \mathbin{\tilde{\sqcup}} (t \mathbin{\tilde{\sqcup}} z))$. Therefore, we define a *generalized*

Algorithm 1 Generalized Join

```
1: procedure $\widetilde{\bigsqcup}(X)$
2:     $(y_j)_{j=1}^{n} \leftarrow sort\_by\_lowerbound(X)$
3:     $z \leftarrow \bot$
4:     for i in (1 ... n) do
5:         $z_i \leftarrow$ reduce(lambda x, y: $\tilde{\sqcup}$(x, y), $(y_j)_{j=i}^{n}$ || $(y_j)_{j=1}^{(i-1)}$)
6:         if $z = \bot$ or $(\#(z_i) < \#(z))$ then
7:             $z \leftarrow z_i$
8:     return $z$
```

pseudo-join operator ($\widetilde{\bigsqcup}$). Given a set of n SASIs, this operator has to produce the SASI z with the least cardinality possible, and such that the n SASIs are included in z. Theoretically, there are $n!$ possible join to consider to pick z. However, as SASIs are traversed clockwise on the number circle, only n of these should be considered. The results of the other $n! - n$ joins are included in one of these n joins. The generalized pseudo-join operator is defined in Algorithm 1, and works as follows: Given a set X of n SASIs, it sorts X elements according to the lexicographic ascending order of their lower bounds (Line 2), producing a new sequence (i.e., $(y_j)_{j=1}^{n}$). Then, referring to the circle number representation, it considers each SASI in $(y_j)_{j=1}^{n}$ and proceeding clockwise joins it with the other SASIs in lexicographical order, producing a final SASI z_i (function *reduce* at Line 5). Finally, the SASI z_i with the least cardinality is returned. The generalized pseudo-join operator is sound by construction, but not monotone. Given three SASIs r, t and z such that $r \sqsubseteq t$, it is *not* always true that $\widetilde{\bigsqcup}(\{r, z\}) \sqsubseteq \widetilde{\bigsqcup}(\{t, z\})$. The lack of the monotone property does not assure termination of the analysis [27], as a least fixed point might not exist. Unfortunately, this property holds for every domain based on number circles.

To address this problem, we defined a *widening* operator to guarantee termination of the analysis. As our widening operator is similar to the one already defined in [27], and for space reasons, it is not presented in this paper.

5 Discussion

As stated in Section 3, our approach is based on the existence of a CFG recovery procedure P_{CFG} that guarantees that all the basic blocks of a function, and its boundary, are retrieved. We do not make any assumption about the capability of P_{CFG} to resolve any indirect jumps, nor to resolve any path predicates. Given such a CFG recovery procedure, our approach can guarantee the soundness of the results.

Though our hypothesis might seem too restrictive in theory, we found it is not to be in practice. In fact, if a binary does not contain data within the boundary of a function f, state-of-the-art CFG recovery procedures, such as [34], can recover every basic blocks and boundary of f precisely. In our experience, most of the employed compilers (e.g., `gcc/g++`) insert data only in specific data sections (e.g., `rodata`). The only exception is represented by jump tables, which might be inserted within a function boundary, thus fooling (in principle) decompilers based on linear sweeping (e.g., objdump [5]).

However, most recent decompilers based on recursive approaches implement algorithms to precisely recover jump tables, and thus, providing in practice the guarantee our approach needs.

6 Evaluation

We run two different evaluations. First, we evaluate the precision of SASI against the related work on signedness-agnostic abstract domains. Then, we implement our static program bloating approach in a tool, called BinTrimmer, and evaluate its efficiency.

6.1 Signedness-Agnostic Strided Intervals

To compare SASI against Wrapped Interval (WI) [27] and quantify its precision, we performed two evaluations using range analyses on both source code and binary files. As shown in the following two sections, on average SASI is 98% more precise than the Wrapped Interval abstract domain.

Source code. For this evaluation, we implemented our Signedness-Agnostic Signed Interval analysis on LLVM and downloaded the publicly-available Wrapped Interval analysis. Then, we retrieved the same test suite utilized by Navas et al. in their work [27]: the SPEC CPU2000 [6]. This dataset is an industry-standardized CPU-intensive benchmark suite, developed from real user applications. As it contains an outstanding amount of mathematical and bitwise operations, it is particularly suited to evaluate abstract domains for numerical entities. Unfortunately, two benchmarks of SPEC CPU2000 (i.e., 300.twolf and 255.vortex) were unavailable at the time of the evaluation. Therefore, we used one more benchmark (462.libquantum) from the latest SPEC CPU (i.e., CPU2006 [7]). Note that we did not use the whole SPEC CPU2006 suite, as it is only available for purchase. Then, we ran the LLVM range analysis [8] on each program in our dataset by using both the SASI and Wrapped Interval domains. For each one of these test, we collected four statistics: The number of variables recovered by using SASI and Wrapped Intervals, which were not $\top$ when the analyses reached a fix-point (indicated as R_{SASI} and R_{WI}, respectively). The number of recovered variables where SASIs provided a better over-approximation (i.e., lower cardinality) than the Wrapped Intervals (indicated as P_{SASI}), and finally the number of variables whose Wrapped Interval representation was more precise than the

[5] https://sourceware.org/binutils/docs/binutils/objdump.html

[6] https://www.spec.org/cpu2000/

[7] https://www.spec.org/cpu2006/

[8] https://code.google.com/archive/p/range-analysis/

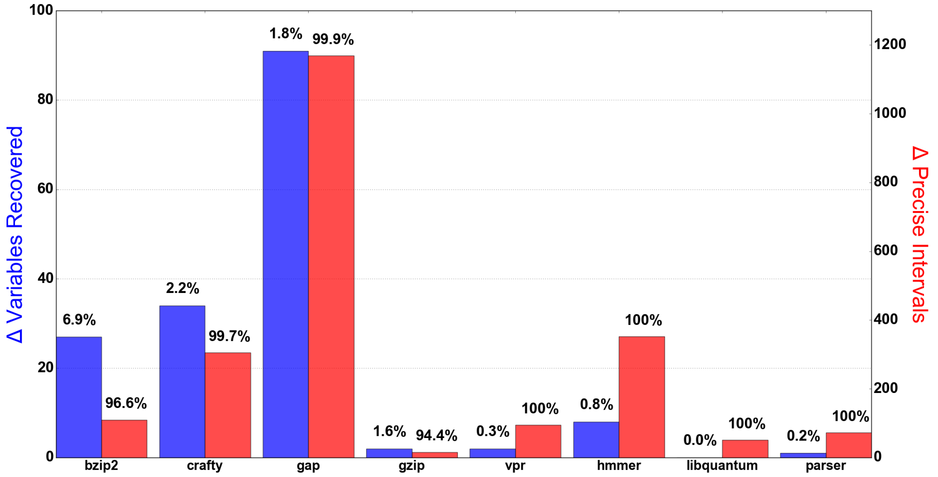


Fig. 3: Source Code Evaluation. *Δ Variables Recovered* indicates the difference between the amount of variables recovered by SASI and Wrapped Intervals. *Δ Precise Intervals* indicates the difference between the number of instances SASI provided a better over-approximation and the number of instances Wrapped Intervals did.

SASI's (indicated as P_{WI}). The results of our evaluation are represented in Figure 3. *Δ Variables Recovered* indicates the difference between R_{SASI} and R_{WI}, and the percentages above each bar quantify the variable recovery effectiveness of SASI (i.e., $\frac{R_{SASI}-R_{WI}}{R_{SASI}\cup R_{WI}}$). *Δ Precise Intervals* indicates the difference between P_{SASI} and P_{WI}), and, similarly, the percentages above each bar quantify the variable recovery *precision* of SASI (i.e., $\frac{P_{SASI}-P_{WI}}{P_{SASI}\cup P_{WI}}$).

As one can see, SASI always recovered more variables than Wrapped Intervals (the Δ of variables recovered is never a negative value), and, in most cases, the variables recovered by SASI were more precise than those recovered by Wrapped Interval. Nonetheless, there were few cases were the variables recovered by Wrapped Intervals were more precise than SASI (e.g., 3.4% in *bzip*2). We investigated them and discovered that it was caused by the lack of associativity of the pseudo-join, as explained in Section 4. In fact, even though our domain's pseudo-join gives more precise results than the Wrapped Intervals' if taken individually, this is not strictly true if we chain them. However, our results clearly show that these cases are rare. For example, SASI recovered 1,170 (out of a total of 5,027) variables in *gap* whose intervals were more precise than those provided by Wrapped Intervals. On the other hand, Wrapped Intervals estimated only one variable in a more precise way than SASI. According to our tests on source codes, we can conclude that, on an average, SASI is 98% more precise than Wrapper Intervals (shown by Δ of precise intervals).

Binary files. To compare SASI's precision against Wrapped Interval's on binary files, we implemented Navas's abstract domain in angr [34]. In this evaluation, we collected all the binaries that DARPA released in the run-up to the CGC final event [9]. Then, we considered the functions of each binary and per-

[9] `http://archive.darpa.mil/cybergrandchallenge/`

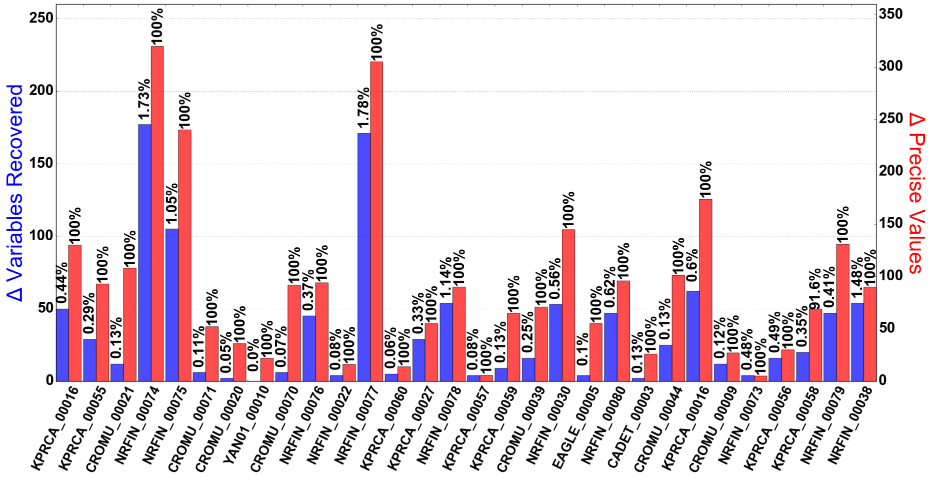


Fig. 4: Binary Evaluation. Δ *Variables Recovered* indicates the difference between the amount of variables recovered by SASI and Wrapped Intervals. Δ *Precise Intervals* indicates the difference between the number of instances SASI provided a better over-approximation and the number of instances Wrapped Intervals did.

formed the angr's value-set analysis on them. Therefore, we collected the SASI and Wrapped Interval representations of each variable (i.e., memory location and CPU register) for each function at each program point, and collected the same four statistics (i.e., R_{SASI}, R_{WI}, P_{SASI} and P_{WI}) already introduced during the source code evaluation. The results collected are depicted in Figure 4.

As one can notice, even in this case SASI always outperformed Wrapped Intervals in terms of variables recovered (the Δ of variables recovered is never a negative value). Furthermore, we noticed that SASI excelled over Wrapped Intervals in terms of precision of recovered variables. In fact, in the case of binaries, SASI succeeded to recovery strictly more *precise* variables (100% values in Δ Precise Values), in every test but once (91.6% success in $KPRCA\_00058$). This result clearly shows the advantage of employing the SASI abstract domain when analyzing binary files.

6.2 BinTrimmer

Our approach to program debloating was implemented in a tool, called BinTrimmer. As introduced in Section 3, BinTrimmer retrieves and patches those basic blocks in a binary that cannot be executed under any execution of a program. Also, in the following we use the term *partial trimming* when a function is partially removed, that is when some function's basic blocks were removed, but not all. BinTrimmer was evaluated by using six binaries linked against two different C libraries: TinyExp [10] (containing 555 LOC) and b64 [11] (containing 192 LOC).

[10] https://github.com/codeplea/tinyexpr

[11] https://github.com/littlstar/b64.c

Table 1: BinTrimmer Results. *Total Trimmed* represents the total amount of code patched, *Min*, *max* and *Avg Partials* indicates the amount of code partially removed with functions. *Gadgets Removed* reports the amounts of ROP gadgets removed, *Tot ICF* is the total number of indirect control-flow transfers, and *ICF Resolved angr* and *ICF Resolved* BinTrimmer indicates the percentage of ICF resolved by angr and BinTrimmer respectively. *Time (min)* shows the time elapsed to analyze the binary.

Program	Total trimmed	Min Partials	Max Partials	Avg Partials	Gadgets Removed	Tot ICF	ICF Resolved angr	ICF Resolved BinTrimmer	Time (min)
TinyExpr1	53.69%	3.62%	83.63%	29.12%	41.3%	2419	99.25%	100%	43
TinyExpr2	7.43%	0%	0%	0%	4.9%	2449	98.65%	100%	87
TinyExpr3	65.67%	3.7%	83.63%	40.33%	56.9%	2419	99.25%	100%	37
b641	1.17%	0%	0%	0%	3.0%	2389	99.6%	100%	24
b642	50.37%	0%	0%	0%	10.6%	2389	99.6%	100%	24
b643	34.43%	1.13%	0%	0%	36.4%	2389	99.6%	100%	22

We dynamically linked both of these libraries to the examples provided on their respective websites, for a total of six different programs.

After running BinTrimmer and removing the identified dead code, we dynamically linked every binary to their patched library, and fuzzed them using *AFL* [12] for 48 hours. No crash was registered. Table 1 summarizes the results of this evaluation. *Total Trimmed* is the percentage of code patched, the *Min Partials*, *Max Partials* and *Avg Partials* values are calculated by considering only those functions that are not completely patched by BinTrimmer. For each of these, we calculate their size (in bytes) and the number of patched bytes and report minimum, maximum, and average values respectively. The *Gadgets Removed* column represents the percentage of ROP gadgets (retrieved with *ROPGadget* [13]) removed by patching each binary's library. The columns *Tot ICF*, *ICF Resolved angr*, and *ICF Resolved* BinTrimmer show the total number of indirect control-flow transfers, the percentage of those resolved by angr alone, and the percentage of indirect control-flow transfer resolved by BinTrimmer, respectively. Finally, we report the *Time* in minutes employed to analyze each program.

Note that, failing to resolve even a single indirect control-flow transfer (i.e., ICF resolved less than 100%) might result in an incomplete CFG, and, therefore, an unsafe program debloating. We also manually checked for each of the six programs the completeness of the recovered CFG: while one CFG contained a super-set of all the possible control-flow transfers (completeness), the remaining five contained all and only the possible control-flow transfers (sound and complete). Note also that BinTrimmer was able to patch code within functions (*Partials* columns). This is an important result as in these cases we outperform even a static linker: To the best of our knowledge, no linker can remove code within functions.

Finally, as we can see from the reported results, there are cases where our approach can remove a conspicuous portion of dead code: in TinyExpr3, we

[12] `http://lcamtuf.coredump.cx/afl/`

[13] `https://github.com/JonathanSalwan/ROPgadget`

Table 2: Comparison of related debloating techniques against BinTrimmer

Technique	Uses Static Analysis	Source Code Not Needed	No Runtime Support	No Testcases Needed
Occam [25]	✓	✗	✓	✗
Chisel [15]	✗	✗	✓	✗
Trimmer [33]	✓	✗	✓	✗
DamGate [5]	✗	✓	✗	✗
Piece-Wise [29]	✓	✗	✗	✓
BinTrimmer	✓	✓	✓	✓

soundly removed 65.67% of the text section, with 40.33% represented by basic blocks within functions.

7 Related Work

Most of the current debloating techniques require test cases for a program to be analyzed. Test cases are used, either statically [25] or dynamically [15,33], to remove code that is not needed for their successful execution. Generally, these techniques are helpful *if* the analyst has a priori knowledge of how the program will be used, which is not true in the general case. Though there exist test cases agnostic debloating techniques, they are highly specialized to specific languages [2,19,20], require the program source code [11,29,36], runtime support [5], or a customized Java virtual machine [39]. Table 2 shows the summary of state-of-the-art debloating techniques on unsafe languages in comparison with BinTrimmer. To our knowledge, BinTrimmer is the first, test case agnostic, static debloating technique that works directly on binaries.

7.1 CFG recovery

BinTrimmer's primary purpose is to recover a complete and precise CFG statically. In the literature, a plethora of work has been done in this direction. Generally, there are two approaches to statically recover a CFG: (i) performing a linear sweep over the target binary, and (ii) performing a recursive disassembly starting from the entry point of the binary.

More advanced disassemblers, decompilers, and binary analysis platforms like IDA Pro, generally follow the latter approach [7,17]. Recursive disassembly produces much better results than linear disassembly, but there are still issues to be solved, the main one being the correct resolution of indirect, or computed branch targets. Failing to resolve targets of an indirect branch entirely will lead to missing code chunks in the recovered CFG. Several approaches [6,32] have been proposed to recover jump tables by performing backward slicing, forward expression substitution, and normal form comparison at the indirect jump site.

Additionally, Kruegel et al. proposed a systematic method [22] consisting of recursive disassembling and statistical analysis to disassemble as much code from obfuscated binaries as possible. Finally, the angr binary analysis framework [34], the framework used by BinTrimmer, uses a combination of the above-said techniques to recover the best effort CFG. While the approaches mentioned thus far are all-traditional-data-flow-analysis-based approaches, Kinder et al. devised a framework, *Jakstab* [21], based on abstract interpretation to recover an over-approximation of control flow graph for binaries.

All the above static CFG recovery techniques still suffer from accurately identifying all the possible targets of indirect control flow instructions (i.e., indirect jumps and calls). BinTrimmer iteratively refines the values of the indirect control flow targets to create a complete and precise CFG.

7.2 Value range analysis

In the literature, there are many examples of such static analyses, including variable bound checking (e.g., to detect buffer overflows) [37,38], detection of logic bugs [10] (e.g., division-by-zero) and various pointer analyses techniques [16,45]. Balakrishnan et al. first proposed a range analysis [1] targeting x86 binaries that can also keep track of the stride. However, strided-intervals require the signedness of the variable and do not take care of the value overflows and underflows. To handle this problem, Navas et al. proposed Wrapped Intervals [27] that is both signedness-agnostic and can take care of the overflows and underflows. However, Wrapped Intervals do not consider the stride and as we show in Section 6 this resolves to less precise results.

8 Conclusions

In this work, we formally presented a new abstract domain called *Signedness-Agnostic Strided Interval* (or *SASI*). SASI is based on the concept of signedness-agnosticism which, together with a careful design of the operations defined on top of it, makes it particularly suited to be used for value set analyses. We evaluated SASI using two different strategies. First, we showed its precision by comparing our results against the related work. Then we showed its potential by presenting a tool for binary analysis, named BinTrimmer, which uses SASI to soundly identify and remove useless code within applications to reduce their possible attack surface. Our implementation of SASI atop both LLVM (for source code analysis) and angr (for binary analysis) is being open sourced to support further research into the field.

9 Acknowledgements

We would like to thank our reviewers for their valuable comments and input to improve our paper. This material is based on research sponsored by the Office of

Naval Research under grant number N00014-17-1-2897, the NSF under Award number CNS-1704253, and the DARPA under agreement number FA8750-15-2-0084 and FA8750-19-C-0003. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, or the U.S. Government.

References

1. G. Balakrishnan and T. Reps, "WYSINWYX: What you see is not what you execute," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, pp. 23:1–23:84, Aug. 2010.
2. S. Bhattacharya, M. G. Nanda, K. Gopinath, and M. Gupta, "Reuse, recycle to de-bloat software," in *Proceedings of the European Conference on Object-oriented Programming*, ser. ECOOP '11, Lancaster, UK, 2011.
3. D. L. Bruening, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, Cambridge, MA, USA, 2004, aAI0807735.
4. J. Caballero and Z. Lin, "Type inference on executables," *ACM Computing Surveys (CSUR)*, vol. 48, no. 4, p. 65, 2016.
5. Y. Chen, T. Lan, and G. Venkataramani, "DamGate: Dynamic adaptive multi-feature gating in program binaries," in *Proceedings of the Workshop on Forming an Ecosystem Around Software Transformation*, ser. FEAST '17, Dallas, Texas, USA, 2017.
6. C. Cifuentes and M. V. Emmerik, "Recovery of jump table case statements from binary code," in *Proceedings of the International Workshop on Program Comprehension*, ser. IWPC '99, Washington, DC, USA, 1999.
7. C. Cifuentes and K. J. Gough, "Decompilation of binary programs," *Software: Practice and Experience*, vol. 25, no. 7, pp. 811–829, 1995.
8. P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL'77, Los Angeles, California, 1977.
9. P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL'78, Tucson, Arizona, 1978.
10. C. Csallner and Y. Smaragdakis, "Check 'n' Crash: Combining static checking and testing," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE'05, St. Louis, MO, USA, 2005.
11. B. Dutertre, A. Gehani, H. Saidi, M. Schäf, and A. Tiwari, "Beyond binary program transformation."
12. J. Feret, "The arithmetic-geometric progression abstract domain," in *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation*, ser. VMCAI'05, Paris, France, 2005.
13. P. Granger, "Static analysis of arithmetical congruences," vol. 30, pp. 165–190, 01 1989.
14. L. C. Harris and B. P. Miller, "Practical analysis of stripped binary code," *SIGARCH Comput. Archit. News*, p. 2005.

15. K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, “Effective program debloating via reinforcement learning,” in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS’18, Toronto, Canada, 2018.
16. M. Hind, “Pointer analysis: Haven’t we solved this problem yet?” in *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE’01, Snowbird, Utah, USA, 2001.
17. N. C. Jain, “Disassembler using high level processor models,” 1999.
18. B. Jeannet and A. Miné, “Apron: A library of numerical abstract domains for static analysis,” in *Proceedings of the International Conference on Computer Aided Verification*, ser. CAV’09, Grenoble, France, 2009.
19. Y. Jiang, Q. Bao, S. Wang, X. Liu, and D. Wu, “RedDroid: Android application redundancy customization based on static analysis,” in *Proceedings of the IEEE International Symposium on Software Reliability Engineering*, ser. ISSRE’18, Berlin, Germany, 2018.
20. Y. Jiang, D. Wu, and P. Liu, “Jred: Program customization and bloatware mitigation based on static analysis,” in *Proceedings of the IEEE Computer Software and Applications Conference*, ser. COMPSAC’16, Atlanta, GA, USA, 2016.
21. J. Kinder, F. Zuleger, and H. Veith, “An abstract interpretation-based framework for control flow reconstruction from binaries,” in *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation*, ser. VMCAI’09, Cascais, Portugal, 2009.
22. C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, “Static disassembly of obfuscated binaries,” in *Proceedings of the USENIX conference on Security Symposium*, ser. SEC’04, San Diego, CA, USA, 2004.
23. W. Landi, “Undecidability of static analysis,” *ACM Letters on Programming Languages and Systems*, vol. 1, no. 4, pp. 323–337, Dec. 1992.
24. J. Lee, T. Avgerinos, and D. Brumley, “TIE: Principled reverse engineering of types in binary programs,” in *Proceedings of the Network and Distributed Systems Security*, ser. NDSS ’11, San Diego, CA, USA, 2011.
25. G. Malecha, A. Gehani, and N. Shankar, “Automated software winnowing,” in *Proceedings of the ACM Symposium on Applied Computing*, ser. SAC’15, 2015.
26. A. Miné, “Abstract domains for bit-level machine integer and floating-point operations,” in *Proceedings of the International Workshop on invariant Generation*, ser. WING’12, Manchester, UK, 2012.
27. J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey, “Signedness-agnostic program analysis: Precise integer bounds for low-level code,” in *Programming Languages and Systems*. Springer, 2012, pp. 115–130.
28. N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’07)*, vol. 42, no. 6, pp. 89–100, Jun. 2007.
29. A. Quach, A. Prakash, and L. K. Yan, “Debloating software through piece-wise compilation and loading,” in *Proceedings of the USENIX Conference on Security Symposium*, ser. SEC’18, Baltimore, MD, USA, 2018.
30. G. Ramalingam, “The undecidability of aliasing,” *ACM Transactions on Programming Languages and Systems (TOPLAS ’94)*, vol. 16, no. 5, pp. 1467–1471, Sep. 1994.
31. S. Sankaranarayanan, F. Ivančić, and A. Gupta, “Program analysis using symbolic ranges,” in *Proceedings of the International Conference on Static Analysis*, ser. SAS’07, Kongens Lyngby, Denmark, 2007.

32. B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited," in *Proceedings of the Working conference on Reverse engineering*, ser. WCRE'02, Richmond, VA, USA, 2002.
33. H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar, "Trimmer: application specialization for code debloating," in *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE'18, Corum, Montpellier, France, 2018.
34. Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) the art of war: Offensive techniques in binary analysis," in *Proceedings of the IEEE Symposium on Security and Privacy*, ser. SP'16, San Jose, CA, USA, 2016.
35. A. Simon, *Value-Range Analysis of C Programs: Towards Proving the Absence of Buffer Overflow Vulnerabilities*, 1st ed. Springer Science & Business Media, 2010.
36. L. Song and X. Xing, "Fine-grained library customization," *arXiv preprint arXiv:1810.11128*, 2018.
37. A. Venet and G. Brat, "Precise and efficient static array bound checking for large embedded C programs," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI'04, Washington DC, USA, 2004.
38. D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," in *Proceedings of the Network and Distributed System Security Symposium*, ser. NDSS'00, San Diego, CA, USA, 2000.
39. G. Wagner, A. Gal, and M. Franz, "'Slimming' a Java virtual machine by way of cold code removal and optimistic partial program loading," *Science of Computer Programming*, vol. 76, no. 11, pp. 1037–1053, 2011.
40. R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making reassembly great again," in *Proceedings of the Network and Distributed System Security Symposium*, ser. NDSS'17, San Diego, CA, USA, 2017.
41. S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling," in *Proceedings of the USENIX Conference on Security Symposium*, ser. SEC'15, Washington, D.C., 2015.
42. H. S. J. Warren, *Hacker's Delight.* Boston, Toronto, London: Addison-Wesley, 2003.
43. D. B. West *et al.*, *Introduction to Graph Theory.* Prentice hall Upper Saddle River, NJ, 1996, vol. 2.
44. L. Xu, F. Sun, and Z. Su, "Constructing precise control flow graphs from binaries," *University of California, Davis, Technical Report*, 2009.
45. S. H. Yong and S. Horwitz, "Protecting C programs from attacks via invalid pointer dereferences," *SIGSOFT Softw. Eng. Notes*, vol. 28, no. 5, pp. 307–316, Sep. 2003.
46. M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar, "A platform for secure static binary instrumentation."

A Signedness-Agnostic Strided Interval Operations

We provided our abstract domain with every mathematical and logical operation included in today architectures' instruction sets. However, due to space constraints, we detail here only the `or` bitwise operation.

Algorithm 2 Bitwise or

```
1:  procedure $|_w$(r, t)
2:      $S \leftarrow \{\}$
3:      for $u = s_u[e, f]w$ in ssplit(r) do
4:          for $v = s_v[g, h]w$ in ssplit(t) do
5:              $t \leftarrow min(ntz(s_u), ntz(s_v))$
6:              $s_z = 2^t$
7:              $m \leftarrow (1 << t) - 1$
8:              $k \leftarrow (e\&m)|(g\&m)$
9:              $u_1 = [(e\& \sim m), (f\& \sim m)]$
10:             $v_1 = [(g\& \sim m), (h\& \sim m)]$
11:             $[lb, ub] \leftarrow u_1 \overset{wr}{|_w} v_1$
12:             $S = S \bigcup \{s_z[((lb\& \sim m)|k), (ub\& \sim m)|k)]\}$
13:     return $\tilde{\bigsqcup}(S)$
```

A.1 Bitwise Or

To define a precise and sound bitwise `or` operation we leverage the *unsigned* version of Warren's algorithm [42], which performs the `or` operation on classic non-wrapping ranges of values. Given two generic SASIs $r = s_r[a, b]w$ and $t = s_t[c, d]w$, the algorithm used to calculate the bitwise `or` operation is shown in Algorithm 2.

First, we split r and t on the south poles, thus avoiding any wrapping intervals (i.e., intervals might include the values 1^w and 0^w). Then, for each u and v resulting from the split, we create a new SASI calculating its stride (s_z) and bounds (lb and ub) separately. For the stride, we retrieve the number of trailing zeros (function ntz) in the bit-vector representations of s_u and s_v both, and consider the minimum of them to set the stride s_z (Lines 5 and 6). In fact, as the strides s_u and s_v have t low-order bits unset, all the values represented by the SASI resulting from $u \ |_w^{wr} \ v$ share the same t low-order bits. Therefore, the choice of a stride equal to 2^t is a sound choice (line 6). The value of these t-lower bits is $k = (e\&m)|(g\&m)$ (where $m = (1 << t) - 1$). On the other hand, the $(w - t)$ high-order bits are handled by masking out the obtained t low-order bits and then applying unsigned version of Warren's `or` algorithm to find the bounds for the SASI resulting from $u|_w v$ (from line 9 to 11). Finally, the SASI resulting from $r \ |_w \ t$ is obtained by applying the generalized join on the list of SASIs collected by applying the algorithm just explained. Since Warren's algorithm employed is sound, the `or` operation is sound.