Analyzing Bytes: Pre-Disassembly Static Binary Analysis

HUAN NGUYEN, Google, USA
SOUMYAKANT PRIYADARSHAN, Stony Brook University, USA
CHENCHENG JIANG, Stony Brook University, USA
R. SEKAR, Stony Brook University, USA

Binary code analysis plays a central role in numerous applications in software security, performance optimization, reverse engineering, and so on. Existing techniques need to first disassemble binaries into functions in assembly code before an analysis can be performed. However, disassembly and function identification have proven to be major challenges for complex variable-length instruction sets such as the x86. A recent trend has been to use static analysis to improve the accuracy of these tasks. This raises a chicken-and-egg problem: a disassembly is needed for static analysis, but a static analysis is needed for accurate disassembly! We overcome this problem by developing a novel static analysis approach that can operate before committing to a disassembly. Our analysis operates on the output of exhaustive disassembly that considers each possible offset in a binary as an instruction, and constructs what is known as a super-set control-flow graph (CFG). The central technical challenge in analyzing this CFG is that it mixes legitimate instructions with unintended ones, causing analysis results from invalid code paths to pollute legitimate ones. To overcome this challenge, we begin with a key new insight that if we focus on backward analyses, we can ensure accuracy of analysis results at intended instructions even though we have no idea where these intended instructions are! Moreover, our analysis operates in time that is linear in the size of the binary. Specifically, in $O(n)$ total time, it yields analysis results for every one of the n offsets in an n-byte binary. For this task, it is orders of magnitude faster than previous techniques, as the previous techniques typically need to repeat the analysis many times.

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

Additional Key Words and Phrases: Binary analysis, Reverse engineering

ACM Reference Format:
Huan Nguyen, Soumyakant Priyadarshan, Chencheng Jiang, and R. Sekar. 2026. Analyzing Bytes: Pre-Disassembly Static Binary Analysis. *Proc. ACM Program. Lang.* 10, PLDI, Article 214 (June 2026), 25 pages. https://doi.org/10.1145/3808292

1 Introduction

Static analysis of binary code plays a central role in software security hardening [4, 11, 24, 25, 32, 42, 47–49, 63, 70, 72, 74, 75, 82, 83, 85, 89, 90, 92, 93], vulnerability discovery [23, 28, 52, 65, 94], malware analysis[15, 40, 87], performance optimization [54, 60], software reliability [53], binary instrumentation [10, 12, 13, 55, 66, 68, 81, 84, 86, 91], and reverse engineering [2, 3, 26, 27, 30, 41, 71, 73, 76, 80].

Binary analysis and instrumentation techniques operate by first disassembling bytes in the binary code into assembly instructions, and then splitting the assembly code into functions. Unfortunately, these are very challenging tasks on most binaries due to their lack of symbol/type information, as well

*This work was supported by an NSF grant 2153056 and an ONR grant N00014-17-1-2891.

Authors' Contact Information: Huan Nguyen, Google, USA, nhuhuan@google.com; Soumyakant Priyadarshan, Stony Brook University, USA, spriyadarsha@cs.stonybrook.edu; Chencheng Jiang, Stony Brook University, USA, chencjiang@cs.stonybrook.edu; R. Sekar, Stony Brook University, USA, sekar@cs.stonybrook.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/6-ART214
https://doi.org/10.1145/3808292

as the complexities of variable-length instruction sets (e.g., x86). These challenges are compounded by compiler optimizations such as function inlining, tail call removal, omission of function prologs, epilogs and base pointers, use of jump tables, and the removal of code following non-returning calls. As a result, state-of-the-art tools are prone to significant disassembly errors [71]. For instance, on the StochFuzz [94] dataset containing binaries with embedded data, Dyninst [35], Angr [80] and Ghidra [76] experienced error rates of 5% to 15% when compiler metadata was disabled [71]. Function identification is even more challenging, leading to error rates of 6% to 25% even on the much simpler Pang et al.'s dataset [61]. Many applications of binary analysis can't tolerate such error rates, as they can lead to crashes of a hardened, optimized or an otherwise improved binary.

Recently, researchers have begun to use static analyses to resolve ambiguities that cause the above-mentioned errors. For instance, accuracy of function entry points can be improved using static analysis to confirm properties such as stack pointer preservation, or more generally, callee-saved register preservation [71, 73]. Detection of function ends can be improved using static analyses to detect tail calls and non-returning calls [27, 55, 71]. Moreover, constructing an accurate CFG requires resolving indirect jump targets through jump table analysis [21, 26, 27, 29, 30, 55, 59, 71, 76, 86, 92]. A static analysis of stack and global memory accesses [5, 19, 31, 42, 43, 67, 77] helps further reduce these errors. State-of-the-art disassemblers such as DASSA [71] and Datalog disassembly (Ddisasm) [29, 30] exploit such static analyses to reduce disassembly error rates by 4× to 10× or more.

Unfortunately, such a use of static analysis leads to a *chicken-and-egg problem:* To begin a static analysis, we need a disassembled binary, but now, this disassembly needs static analysis results! In practice, circularity is addressed using an iterative approach: a *speculative* initial disassembly phase considers multiple interpretations of a binary's bytes, followed by a static analysis to eliminate all but one of these interpretations. Since static analysis techniques tend to operate on functions, systems such as DASSA [71] may need to consider every byte offset in a binary as a potential function start. This means that in the place of analyzing a single function of size n, they may analyze $O(n)$ functions, each starting at one of the n offsets. As a result, even a linear-time static analysis techniques may end up taking quadratic time due to the $O(n)$ repetitions of the same analysis. In practice, researchers use heuristics and techniques to reduce the number of static analysis invocations. Despite these efforts, a super-linear slowdown is evident in the performance results of DASSA, with per-instruction analysis time increasing from about 0.3 milliseconds to 2 milliseconds, as the binary size increases from about 10K to 500K instructions. (See Fig. 15 in [71].) The challenge is more pronounced for Ddisasm, with some binaries (e.g., *wrf* from the SPEC benchmark suite) failing to disassemble due to resource exhaustion.

In contrast, we propose a novel approach to solve this chicken-and-egg problem by *analyzing the bytes in a binary before committing to a specific disassembly.* In time ***linear in the binary size***, our analysis produces results at ***every byte offset in the binary,*** with accuracy comparable to previous works.

1.1 Background, Challenges and Key Insight

Since instruction behaviors aren't apparent from numeric opcodes, our first step is to decode them into assembly instructions. Rather than committing to a specific interpretation of the binary, *exhaustive disassembly* [50] considers every byte offset as the possible start of an instruction, and decodes it. The result will include all *intended* instruction offsets (as given by the correct/intended disassembly of the binary), as well as unintended ones. Note that many of the decoded instructions are overlapping: a k-byte instruction starting at offset l can overlap with $k-1$ instructions obtained by decoding offsets $l+1$, $l+2$,...,$l+k-1$. Neither hand-coded assembly nor compiler-produced code relies on overlapping instructions, so at most one of these k instructions is intended.

Since binary analysis tools don't know the "intended" disassembly, they work with a subset of instructions produced by exhaustive disassembly. We will assume that this subset is *control-flow consistent,* i.e., the direct successor(s) of every instruction is present in the disassembly. A non-control

flow instruction of size k at offset l has a single successor, located at $l+k$. An unconditional control transfer instruction also has a single successor whose location is specified in the instruction itself. Finally, a conditional branch combines the two cases and hence has two possible successors.[1]

Although the number of such disassemblies is exponentially large, Krugel et al. [50] showed that they can all be represented in a single control-flow graph (CFG) that is *linear* in the binary size. Each node in this *superset CFG* corresponds to one of the n offsets in the binary, and contains the instruction decoded from that offset. Edges in the graph connect successor nodes. Since the maximum size of each instruction is bounded by a constant and since the number of edges per node is at most two, it is clear that the size of the superset CFG is $O(n)$, where n denotes the binary size.[2]

Superset CFG presents a natural starting point for binary analysis as well as instrumentation. Indeed, Multiverse [10] is an innovative approach for instrumenting the superset CFG. Instrumented binaries are linear in the size of the original binary, and ensure correct runtime behavior regardless of which subset of instructions are the intended ones. However, no similar techniques have been developed for static analysis. This is because instrumentation is simpler in one key aspect: *it is not necessary to predict all possible execution paths that may have been followed in the past, or will be taken in the future.* Instrumentation correctness only requires that *as and when an instruction is executed,* its associated instrumentation is also executed.

In contrast, static analyses generally reason about all-path properties and hence need an approximation of all possible paths in the program. One can of course use the superset CFG to provide this approximation, but this can lead to catastrophic imprecision, since *static analysis results at intended instructions may be polluted by meaningless results flowing from non-instructions.* To illustrate this, consider offsets $l-k_1,l-k_2,...l-k_r$ such that the decoded instruction at $l-k_i$ is of length k_i. At most one of these instructions is intended, yet a dataflow analysis using the superset CFG will need to consider the effect of all r instructions when computing the state at location l. Note that the state computed at unintended instructions is likely to be meaningless or $\top$, and when it is combined with the state flowing from valid instructions, the result is apt to become $\top$ and thus lose all precision. Moreover, imprecision compounds with each step, so even if precision isn't lost in the first step, it will soon be lost, leading to results that are too imprecise to be useful.

Challenge 1: Separating wheat from chaff — without knowing which is which.

Our first question is whether we can separate the true analysis results at the intended instructions ("wheat") from meaningless results gathered at unintended instructions ("chaff"). Unfortunately, we don't even know where the intended instructions are. To determine if such separation is feasible, we first make a note about what causes the mixing described above: control-flow consistency implies that the *successor of an intended instruction will always be an intended instruction,* but it does not guarantee the converse. As such, the predecessor of an intended instruction may be an unintended instruction, which allows analysis results computed for unintended instructions to pollute those of intended instructions. The following insight enables us to overcome this challenge:

Key Insight 1 (Backward Path Fidelity). *Any instruction that is backward reachable from an unintended instruction will itself be unintended.*

Proof: For any two locations x and y in the binary, control flow consistency means

$$succ(x,y) \land intended(x) \rightarrow intended(y)$$

where the notation $succ(x,y)$ denotes that y is the successor of x, i.e., there is a forward edge in the superset CFG from x to y. (Note that $pred$ denotes the inverse of $succ$, i.e., it denotes a backward edge

[1] Control-flow consistency also means that there are no illegal instructions. Many disassembly techniques also assume non-overlapping instructions, but this isn't strictly necessary in our approach.

[2] Indirect branches are not represented at this point because we have no information about their possible successors.

in the superset CFG.) Taking the contrapositive of this implication, we get

$$unintended(y) \rightarrow unintended(x) \vee \neg succ(x,y).$$

Addition of the condition $succ(x,y)$ to both sides of this implication yields

$$succ(x,y) \wedge unintended(y) \rightarrow unintended(x),$$

which is the same as $pred(y,x) \wedge unintended(y) \rightarrow unintended(x)$, i.e., every predecessor of an unintended instruction will itself be an unintended instruction. ■

Corollary. *Intended* instructions are *unreachable* in the backward direction from *unintended* instructions. Consequently, analysis results computed at unintended instructions will not pollute those of valid instructions during a *backward analysis*.

This seems magical: we don't know what the intended instructions are, yet we can be sure that the static analysis results computed at those instructions won't be affected by unintended instructions!

Challenge 2: Computing path properties without knowing where the paths begin.

Properties of interest are typically path properties, e.g., the preservation of stack pointer on a path from a function entry to its exit. In a conventional static analysis setting, the starting point for the analysis is identifiable because we would already know the entry point of functions. However, in our setting, we start with a superset CFG, so there is no clear starting point. We cannot speculatively try various offsets in the binary as starting points, as our entire goal is about avoiding the quadratic runtime complexity of such an approach.

To address this problem, we leverage our focus on intra-procedural backward analysis. Since functions end with a return instruction,[3] they serve as a natural starting point for our analysis. However, we don't yet know the *intended* return instructions, so we are not home free (yet).[4]

Our solution to this problem is to start the analysis from *every* return instruction in the superset CFG, and propagate information backwards. For the stack pointer (SP) preservation property, our goal is to compute, for each offset in the binary, the difference between the SP value at this offset and the previous return instruction on the path. We refer to this quantity as the *stack depth.* Backward path fidelity ensures that if this is an intended instruction offset, then it would only contain the stack depth with respect to an intended return instruction. Moreover, there cannot be multiple valid return instructions on a valid backward path, so the computed quantity will be the stack depth with respect to the unique valid return instruction encountered on this path.

Challenge 3: Achieving linear-time complexity.

In a conventional analysis that processes a single function at a time, non-linear runtime complexity can be acceptable, especially for functions that aren't too large. But in the context of an analysis that operates on the entire superset CFG, nonlinear complexities can be unacceptably expensive.

To achieve linear-time complexity while providing results at every byte offset in a binary, an analysis needs to satisfy the following conditions:

- It computes a constant amount of information (i.e., $O(1)$) at each offset. For analyses that compute path properties, this means that we compute a single property common to all paths.
- It should take only a constant amount of time to compute results at an offset x from the results at the successors of x (of which there can be at most two in the superset CFG).
- Results should be computed in one shot, and should not involve any looping of previous steps.

[3]We discuss unusual cases (e.g., functions ending with a non-returning call, tail calls) in a later section.
[4]Return is a single-byte instruction on x86, but that does not help distinguish *intended* return instructions from unintended ones.

1.2 Overview and Paper Organization

Given the benefits of backward analysis on superset CFG, the rest of this paper is geared towards backward analyses. We prioritize analyses that help to identify unintended instructions: once (most of) the unintended instructions are eliminated, any combination of forward and backward analyses can be used without undue precision loss. Such combinations are more versatile and can compute a far broader class of properties than what backward analysis can do on its own.

Specifically, we focus on disambiguating per-function CFGs through accurate identification of function starts and function ends. Once this is done, we can construct an accurate CFG for that function consisting of intended instructions. Our choice is also guided by a desire to compare the accuracy of our analysis with previous research. As a result, we focus on the following analyses:

- *Stack depth:* Stack depth is defined as the difference between the stack pointer value at the current instruction and the last return instruction that was examined in the current backward path. An instruction with a stack depth of zero is consistent with function starts, and is taken as evidence that the current instruction is a function start.
- *Register preservation:* For each instruction offset, this analysis computes the set of registers preserved *actively* or *passively* on the backward paths from a return to this instruction offset. This analysis is also aimed at function starts, as they should preserve all callee-saved registers.
- *Non-returning calls:* This analysis determines if a given offset contains a call that will never return.
- *Invalid pointer dereference:* This forward analysis was proposed in [71] as a way to flag data bytes in a binary. We present a backward version of this analysis that works on the superset CFG.

1.3 Contributions

- We propose AMACE (Analyzing Machine Code Efficiently), the first static analysis method to operate directly on binary code, before committing to any particular disassembly of these bytes. In doing so, it breaks the circular dependency between static analysis and disassembly.
- On the analysis tasks AMACE has been applied to, it outperforms previous forward analysis techniques, specifically, those developed in DASSA [71]. In addition to being orders of magnitude faster, we present additional techniques in our analysis that achieve increased accuracy over DASSA.
- We apply our static analysis results to address two important problems in reverse engineering: detecting function entries and non-returning calls. Our F1-scores of 94% and 73% are on par with state-of-art tools, demonstrating that our performance benefits are achieved without sacrificing accuracy or generality.
- Our techniques scale linearly with program size. In particular, after $O(n)$ processing time, our analysis produces results for each and every one of the n offsets in the binary.

2 Approach Overview and Illustration Using Invalid Pointer Dereference

Invalid pointer dereference analysis was proposed by Priyadarshan et al. [71] to flag data or unaligned code — *unintended code* in our terminology. The conceptual simplicity of this analysis makes it an ideal candidate for introducing our backward analysis approach.

Intended code is *likely* to satisfy many properties such as define-before-use. But we need to be mindful of the fact that intended code can still be buggy — buggy code should not be classified as invalid code. As such, Priyadarshan et al. [71] suggest the use of properties whose violation is likely to cause a serious runtime failure, such as a crash. Uninitialized pointer dereferencing is one such error. They showed that when combined with other techniques, it can cut down the false positives by 2× or more, while increasing false negatives by just 0.01%.

While Priyadarshan et al. present a forward analysis that occurs after disassembly, we develop a similar *backward analysis before committing to a specific disassembly.* Moreover, our analysis proceeds in a single pass over the entire binary, computing static analysis results at every offset in the binary. Producing a similar result using Priyadarshan et al.'s DASSA approach [71] will require their analysis to be repeated at every offset in the binary, which is very expensive. (See §6.2.)

2.1 Superset CFG Construction and Lifting to an IR

Our first step that precedes any analysis is the construction of the superset CFG. We create a graph node for each offset in the binary. The instruction at the offset is disassembled, and a forward edge introduced from this node to the node at its successor location. A backward edge from the successor to the current node is also added. One special case is a call node, which introduces a forward edge to the target, but there is no backward edge since the target cannot be determined statically. Finally, if an offset doesn't contain a valid instruction, then it has no forward edge. In addition, we traverse backward edges from such instructions and delete the ancestors reached, until we encounter an ancestor that has at least one valid successor. By the backward path fidelity property, the instructions deleted in this manner must all be unintended instructions.

Next, we compute the strongly connected components (SCCs), which occur due to loops in the program. Within loops, we cannot always ensure that we have analyzed the predecessor of an instruction before analyzing the instruction itself. But outside of SCCs, we use a reverse post-order (RPO) traversal to ensure this property. (Since we are performing a backward analysis, the notions of SCC and traversal order are with respect to the backward edges.)

Disassembled instructions are lifted to a simple intermediate representation (IR) that is based on an architecture-neutral, register-based core intermediate language shown in Fig. 1. Our implementation handles a slightly more general language, but the core language in Fig. 1 is sufficient to capture all of the key elements of our static analysis. For lifting, we rely on the LISC system by Hasabnis et al. [36–38].

Indirect control transfers are unresolved at this point, and we leave it that way for the rest of the paper. While it may seem that this reduces analysis accuracy, we point out this is standard practice in binary analyses that operate at disassembly time [7, 30, 56, 71]. Specifically, most tools, including ours, perform intra-procedural analysis where call targets (including those of direct calls) and return targets aren't used. However, intra-procedural indirect transfers can occur due to jump tables. [5]

Because indirect targets are unresolved, we design our analysis techniques to cope with them. In particular, indirect calls and returns don't pose a problem, as already noted above. Direct tail calls are simply direct jumps, so they don't pose a problem either. Indirect tail calls and indirect jumps do occur, but are infrequent. Although our analyses can't derive useful information about them, they compensate by using information from other program paths that are free of them. For instance, jump table code typically contains a bounds check before accessing the table, with the failure branch being a direct branch to the default case. Since most of our analyses look for a program path that violates a certain property, e.g., stack pointer preservation, soundness isn't compromised by ignoring paths with indirect transfers.

2.2 Invalid Pointer Dereferencing in a Backward Analysis Setting

In general, a forward binary analysis can more accurately reason about memory contents as compared to backward analysis. Techniques such as value set analysis (VSA) [5] were developed as a forward analysis for this reason. In a backward analysis, when we reach an instruction dereferencing a memory location, we typically haven't yet analyzed the code that wrote into that location. For this

[5]Jump tables typically result when large switch statements are compiled into binary code. The switch statement is translated into an indirect jump that obtains its target address from a lookup table at runtime.

$$
\begin{array}{rcll}
O & ::= & R_i \mid c & \text{(Operands: registers or constants)} \\
T & ::= & R_i \mid *(O{+}c) & \text{(Assignment target: a register or memory)} \\
E & ::= & O \mid *(O{+}c) \mid O{+}O & \text{(Expressions. ``+'' stands for a generic arithmetic operator)} \\
S & ::= & T = E & \text{(Assignment)} \\
 & \mid & \texttt{call}\ O \mid \texttt{ret} & \text{(Call and return)} \\
 & \mid & \texttt{br}\ O & \text{(Unconditional branch)} \\
 & \mid & \texttt{cb}\ c & \text{(Direct conditional branch)} \\
 & \mid & \texttt{end} & \text{(Instructions that cause execution to end)}
\end{array}
$$

Fig. 1. Core language (IR) that is the input to our analysis.

reason, we simplify our analysis to consider only the register contents. Secondly, we structure our analysis to gather *constraints* on register content rather than actual values. This constraint form is more suitable in a backward setting. In particular, whenever a register value is used as a memory address, we add the constraint that this register should be initialized.

We use the notation $def(R)$ to denote the constraint that R should be defined. Note that indirect calls and jumps specify their target in a register, causing it to be dereferenced when fetching the instruction at the target. Thus, they too require the register to be defined.

If a register R needing to be defined is computed from other registers, then this computation satisfies $def(R)$ constraint, provided the registers on the right-hand side are themselves defined. So we add $def(X)$ for every register X used on the rhs. Moreover, a call sets the values of registers that are designated to hold return values as per the ABI (e.g., registers `rax` and `rdx` on x86-64/Linux).

The results of this analysis can help answer disassembly-related questions. Specifically, to determine if offset l could be a function start, we check if the set $\{R \mid def(R)\}$ is a subset of registers whose values are specified by the ABI (application-binary interface) at function starts, e.g., registers used for argument passing and the stack pointer (SP). If not, we say that l is a non-function.

2.3 Equations specifying Invalid Pointer Dereferencing Analysis

We formulate the problem as a backward dataflow analysis, with $in(S)$ representing definedness requirements just after the statement S; and $out(S)$ representing the requirements just before S. Equations for computing *out* from *in* are shown in Table. 1. Line 1 in the table handles the base case of instructions that have no successors. Nothing needs to be defined at these instructions, so the set of registers needing to be defined is empty. Line 2 encompasses all instructions that dereference a memory location derived from the contents of a register. Given our language (Fig. 1), this means that the memory address is of the form R or $R{+}c$ for some register R. (This characterization includes instructions for reading data as well as indirect jumps/calls.)

Lines 3 and 4 consider instructions that compute a register R_i from other registers. If $in(S)$ contains the condition that R_i be defined, then this definition satisfies the condition. But in order for R_i to be defined, the registers used to define it should themselves be defined. Line 5 considers register assignments where the right-hand side is more complex — as per our language, this corresponds to

#	S	$out(S)$	Comments
1	`ret`, `end`	$\{\}$	Nothing needs to be defined at return or end.
2	Dereferences R	Add $def(R)$	R must be defined before its use in memory dereferencing or control transfer target.
3	$R_i = R_j + R_k$	Repl. $def(R_i)$ by $def(R_j), def(R_k)$	Assigning to R satisfies $def(R)$, so it can be removed. But the registers needed for this will have to be marked as needed.
4	$R_i = R_j + c$	Repl. $def(R_i)$ by $def(R_j)$	
5	$R_i = E$	Delete $def(R_i)$	
6	`call`	Delete $def(R)$ for $R \in$ return_regs	Return value registers will be defined after calls.
7	cb O	$def(R_i)$ if $def(R_i)$ is in $in_1(S), in_2(S)$	Intersect at joins.
8	Any other	$in(S)$	Other operations don't affect def.

Table 1. Equations defining uninitialized pointer dereference analysis.

reads from memory. In this case, we discharge the condition that R_i be defined, but don't propagate it to the memory operand because this analysis does not track memory contents. (However, note that the memory address being dereferenced is still subject to the definition on Line 2.)

Line 6 specifies that return registers are defined after a call. (If the call is indirect, it is still subject to the equation on Line 2.) Line 7 handles conditional branches, which are the only instructions requiring a join in our analysis. Since we cannot predict which of the branches may be taken at runtime, we conservatively say that a register must be defined if both branches require it to be defined. Line 8 handles all other cases, where we carry over $in(S)$ to $out(S)$.

2.4 Illustration with an Example

Fig. 2 shows an example snippet and our analysis results on the snippet. Note that the figure shows the disassembly at every byte offset, so it contains a combination of intended and unintended instructions. Intended instructions are shown in bold face. In each line, we use the *succ* column to show the line number of the next instruction — this will always be an intended instruction for an intended instruction. For an unintended instruction, its successor can either be intended or unintended. In order to illustrate these properties more clearly, we have left the example in binary code and assembly code format rather than the IR used in our analysis. Since the equations in Table 1 rely on properties such as the registers dereferenced by an instruction, they can as easily be applied to assembly instructions.

The analysis begins at the return instruction. Because the successors of the unintended instruction at `0x12` are omitted, we conservatively assume its *in* set is empty, yielding an empty *out* (Table 1, Line 8). The same rule applies to `0x11`, whose *in* is also empty. At `0x10`, the instruction dereferences `rcx`, so it introduces the constraint `def(rcx)` (Line 2, Table 1). Continuing similarly, the resulting constraints appear in Table 2. For illustration, the instruction at `0x0b` dereferences a pointer derived from `rdi` and `rax`, requiring both to be defined, and it also propagates the incoming `def(rcx)` from its successor at `0x0e`. At `0x05`, a value is loaded into `rax`, clearing its definition requirement. Finally, one successor of the conditional branch at `0x03` is omitted; we therefore treat its *in* as empty. By the intersection-based join rule, no register definitions are required at this point.

Based on the *def* constraint at each offset, we can rule out many of them for function starts. In particular, any instruction with $def(rax)$ cannot be a function start because the value of `rax` register is not defined at function start. Note that `0x00` can be a function start because there is no constraint on

Loc	Succ	Opcode	Mnemonic	Constraint/Action
0x00	**03**	**83 FF 74**	**cmpl $0x74, %edi**	
0x01	05	FF 74 74 11	pushq 0x11(%rsp, %rsi, 2)	def(rsi)
0x02	04	74 74	je 0x78	clear
0x03	**05**	**74 11**	**je 0x16**	clear
0x04	07	11 48 8B	adcl %ecx, -0x75(%rax)	def(rax)
0x05	**0c**	**48 8B 05 19 B0 18 00**	**movq 0x18b019(%rip), %rax**	clear rax
0x06	0c	8B 05 19 B0 18 00	movl 0x18b019(%rip), %eax	clear rax
0x07	0c	05 19 B0 18 00	addl $0x18b019, %eax	def(rax)
0x08	0e	19 B0 18 00 64 C7	sbbl %esi, -0x389bffe8(%rax)	def(rax), def(rcx)
0x09	0b	B0 18	movb $0x18, %al	def(rdi), def(rax), def(rcx)
0x0a	0c	18 00	sbbb %al, (%rax)	def(rax)
0x0b	0e	00 64 C7 00	addb %ah, (%rdi, %rax, 8)	def(rdi), def(rax), def(rcx)
0x0c	**13**	**64 C7 00 31 31 31 C0**	**movl $0xc0313131, %fs(%rax)**	def(rax)
0x0d	13	C7 00 31 31 31 C0	movl $0xc0313131, (%rax)	def(rax)
0x0e	10	00 31	addb %dh, (%rcx)	def(rcx)
0x0f	11	31 31	xorl %esi, (%rcx)	def(rcx)
0x10	12	31 31	xorl %esi, (%rcx)	def(rcx)
0x11	13	31 C0	xorl %eax, %eax	
0x12	15	C0 C3 90	rolb $0x90, %bl	
0x13	**14**	**C3**	**retq**	

Table 2. Illustration of invalid pointer dereferencing analysis. Intended instructions are in bold face.

rax. This is because rax is first initialized at 0x05 before it is dereferenced. In addition, instructions that have constraints only on the registers used for passing arguments — specifically rcx, rdi and rsi in this example — can also be function starts. In sum, the analysis rules out about half of the locations in this example as possible function starts. While this may seem modest, this is consistent with the numbers reported in previous work [71], where its addition cut down the false positive by about 2× without increasing false negatives.

3 Stack Depth Analysis

Stack depth analysis was proposed as a technique for function identification by Rui et al. [73]. The underlying capability to analyze register values has been part of many previous binary analysis techniques as well [5, 77]. Our key contribution here falls into two main categories. First, we show that it can achieve good accuracy before committing to a particular disassembly of a binary. Second, our analysis develops measures to improve its accuracy in the face of non-returning calls. In contrast, previous techniques are prone to false negatives due to unrecognized non-returning calls. (See §3.3 for more explanation.)

3.1 Backward Analysis Formulation and Description

Similar to invalid pointer dereferencing, the goal of stack depth analysis is to identify *definite non-functions,* i.e., locations that can be ruled out as function starts. Towards this goal, the analysis computes $SP_R - SP$ at each offset l in the binary, with SP and SP_R representing the stack pointer values at l and the return instruction (encountered on the backward path reaching l) respectively.

To support better decisions, our analysis computes triples of the form $\langle d, d_{max}, r \rangle$ at each location l. Here, d denotes the stack depth at l, d_{max} denotes the maximum stack depth seen on the backward path reaching l, and r denotes the number of times this observation has been made. Note that r loosely relates to the number of backward paths on which this observation has been made. Both d and d_{max} can have any integer value. In addition, d can have two special values: $\top$ indicates an unknown value, while "?" indicates one of many integer values. We use "?" instead of remembering the specific values in order to limit memory usage and analysis complexity.

Some functions may contain paths that don't conserve stack depth, e.g., a path that ends in an (undetected) non-returning call. Static analyses will treat the code following such calls as part of the same control-flow path, which will lead to an incorrect estimation of stack depth. To handle such errors, our analysis doesn't insist on a unique stack depth at each location but allows for *a set* of triples: one corresponding to each group of paths that agree on the stack depth. To ensure linear-time complexity, the size of this set is capped at a small constant k. (If more than k distinct values arise in an analysis step, some of them will be merged into one.) An empty set at an offset l indicates one of three possibilities:

(i) l in on a known non-returning path, i.e., ends with a prohibited instruction, (e.g., hlt) or a call to a well-known non-returning function (e.g., exit);
(ii) l has no known successors, i.e, l is an indirect branch instruction; or
(iii) l hasn't yet been analyzed. (This can happen when analyzing a loop.)

Our analysis is given by the dataflow equations in Table 3. For each statement S in the IR, if $in(S)$ is the stack depth immediately after executing S, then $out(S)$ specifies the depth just before executing S. The first two lines in the table correspond to base cases, i.e., instructions that end a forward path (and are hence the beginning of a backward path). Line 3 handles instructions that increment or decrement the stack pointer by an amount specified in the instruction. Our analysis increments or decrements the stack depth by the same amount. But if the stack is adjusted by an amount that cannot be determined from the instruction, we handle it conservatively by setting stack depth to $\top$, representing an unknown depth. Most other types of instructions (including assignments to registers

#	S	$out(S)$	Conditions/Comments
1	`ret`	$\{\langle 0,0,1\rangle\}$	By definition, stack depth at return is zero.
2	`end`, `br` R	$\{\}$	As per (i) and (ii) below.
3	$SP{=}SP{+}c$	$in(S){+}c$	Add c (positive or negative) to all depth components in $in(S)$.
4	$SP{=}E$	$\{\langle\top,\top,1\rangle\}$	Complex assignment to SP that we can't reason about.
5	$T{=}E$	$in(S)$	Stack depth unaffected by assignments to memory or other registers.
6	`call` O	$in(S)$	Calls are expected to preserve SP.
7	`br` O	$in(S)$	Branches don't change SP.
8	`cb` O	$in_1(S)\sqcup in_2(S)$	Merge info from the two successors.

Table 3. Equations defining stack depth analysis.

other than SP, calls and unconditional control transfers) do not affect the stack depth, and this is captured by Lines 5–7. Finally, conditional branches are handled on Line 8 by merging $in(S_1)$ and $in(S_2)$, where S_1 and S_2 are the two successors of this instruction. The merge operation ("$\sqcup$") works as follows. It first merges triples from in_1 and in_2 that have identical depths

$$\langle d,d_{max1},r_1\rangle\sqcup\langle d,d_{max2},r_2\rangle=\langle d,max(d_{max1},d_{max2}),r_1+r_2\rangle$$

If the depth of a triple in in_1 does not match that of any in in_2 (or vice-versa), that triple is added as-is to the output set. Let s be the size of the output set at this point. If $s\le k$, we are done. Otherwise, $s-k+1$ of the triples with non-$\top$ depth and the lowest r values are merged into a single triple $\langle ?,d_{mm},r_s\rangle$, where d_{mm} is the maximum of the d_{max} values being merged, and r_s is the sum of the r values being merged. (If there is already a tuple with the source "?" then $s-k$ of the tuples are merged into that tuple.)

3.2 Illustration with an Example

Table 4 demonstrates our analysis on a simple function. Unlike earlier examples, we omit machine code and superset disassembly. The backward path-fidelity property ensures that unintended instructions can be ignored: their results never affect intended instructions, allowing us to dispense with low-level byte code details.

The function in Table 4 begins at L0 and has two return paths, one of them ending with an *unknown* non-returning call (to nn@plt at location L1). The analysis starts at the return instructions and proceeds backward, with the applied equations from Table 3 shown in the "Explanation" column. Irrelevant instructions are elided. There are two merge points. At `jg L2`, both successors have the same depth, yielding a single merged tuple. At `je L1`, the successors differ, producing a tuple with two sets. At L0, most paths have depth zero, providing evidence that L0 is a function entry.

Lbl	Instruction	IR (S)	$out(S)$	Explanation
L0	...	$\cdots$	$\{\langle 0,48,2\rangle,\langle 16,48,1\rangle\}$	
	je L1	cb L1	$\{\langle 0,48,2\rangle,\langle 16,48,1\rangle\}$	(8)
	push %rbx	rsp=rsp-8, *(rsp)=rbx	$\{\langle 0,48,2\rangle\}$	(3)
	push %rsi	rsp=rsp-8, *(rsp)=rsi	$\{\langle 8,48,2\rangle\}$	(3)
	jg L2	cb L2	$\{\langle 16,48,2\rangle\}$	(8)
	...	$\cdots$	$\{\langle 16,16,1\rangle\}$	
	pop %rsi	rsi=*(rsp), rsp=rsp+8	$\{\langle 16,16,1\rangle\}$	(3)
	pop %rbx	rbx=*(rsp), rsp=rsp+8	$\{\langle 8,8,1\rangle\}$	(3)
	ret	ret	$\{\langle 0,0,1\rangle\}$	(1)
L1	call nn@plt	call nn@plt	$\{\langle 16,48,1\rangle\}$	(6)
L2	sub $32, %rsp	rsp=rsp-32	$\{\langle 16,48,1\rangle\}$	(3)
	...	$\cdots$	$\{\langle 48,48,1\rangle\}$	
	add $48, %rsp	rsp=rsp+48	$\{\langle 48,48,1\rangle\}$	(3)
	ret		$\{\langle 0,0,1\rangle\}$	(1)

Table 4. Stack depth analysis illustration on an example function starting at L0. It has three paths, two ending on `ret`s, and one on a non-returning call at L1. The last column shows the applicable equations from Table 3.

3.3 Soundness and Precision

Stack depth analysis collects evidence that the code beginning at an offset l *does not* conserve the stack. Approximations are designed to discard existing evidence, but won't create evidence where none exists. Even so, there are some assumptions being made that can affect soundness. The most important case, from a practical perspective, is that of unrecognized non-returning calls. The analysis considers the invalid path that returns from such a call, and computes the corresponding (incorrect) stack depth. This depth will likely not match those on valid paths, leading to a potential false negative (and hence a soundness bug). This problem can be mitigated using a thresholding scheme if non-returning calls are infrequent. Specifically, for a location l, if a majority of the paths in the analysis output have a stack depth of zero, then we return an "yes" to the question of whether l could be the start of a function. From a practical perspective, the biggest source of precision loss is due to variable sized arrays, which will cause the stack pointer to be changed by a non-constant value. The analysis conservatively sets the stack depth to be $\top$, which can lead to a non-function to be flagged as a potential function. We address this by reasoning about the base pointer (BP) in addition to the SP. We omit the details here since it is similar to callee-saved register preservation discussed in the next section.

4 Callee-Saved Register Preservation

Callee-saved register preservation has been used successfully in previous work for function entry identification [71, 73]. Preserved registers are either left unused in a function, or saved to memory and later restored. Since previous techniques used forward analysis, they were capable of modeling memory, e.g., using abstract interpretation [5, 22, 71, 77]; and check if the restored value is the same as what was stored. In contrast, a purely backward analysis makes memory modeling difficult: memory accesses use addresses stored in a register, but we cannot determine this register's value from the backward path reaching this instruction. To address this challenge, we rely on two observations: (a) stack memory is used for saving and restoring registers, and (b) stack addresses are typically constant offsets from the current SP. We leverage these observations to track register saves/restores in most cases.

Conceptually, our analysis follows backward through the chain of assignments that produce a callee-saved register's value at a return instruction. For each offset l backwards-reached from a return, we abstract this chain using a *restore summary* of the form: $S \rightarrow I \rightarrow D$. It states that the value of the source operand S at location l will be the one stored in the callee-saved register D at the return instruction. The source S is typically any register, but can also be $\top$. The optional intermediary I represents the last memory location along this assignment chain. Since we track only stack memory, this location must have the form $*(SP{+}c)$. We may abbreviate $S \rightarrow I \rightarrow R$ by $S \rightarrow R$ in some contexts.

As with stack-depth analysis, approximations made in register preservation are also biased towards soundness: specifically, the analysis is designed to report when a register is *definitely not preserved.* Note that a restore summary of $\top \rightarrow X$ means that X *may not* be preserved, so we won't consider l as non-preserving just because of this summary. Instead, we require *evidence of mismatch* in order to classify l as a non-function. Examples of mismatch include summaries of the form $R' \rightarrow R$ and $*(SP{+}c) \rightarrow R$. The former indicates a restore from an incorrect register, while the latter indicates a restore from a memory location to which no store has been made.

Despite approximations oriented for soundness, unrecognized non-returning calls violate our assumptions, and can lead to soundness violations. To minimize their impact, we declare an offset l as a non-preserving only if a majority of paths indicate that registers are not preserved.

4.1 Analysis Description

Our register preservation analysis is defined by the equations in Fig. 5, with CSR denoting the set of callee-saved registers. Step 1 handles the base cases. Step 2 is applicable at or before the last instruction

Condition	S	$out(S)$	Comments
1. Initialize			
	`ret`	$\{R \to R \mid R \in CSR\}$	Each register contains its own value at return.
	`end`, `br` R	$\{\}$	Non-returning path, so nothing is restored.
2. Construct the chain from return to the intermediary (saved location on the stack)			
$in(S)$ contains $R' \to R$	$R' = *(SP+c)$	Repl. by $*(SP+c) \to R$	Process the last assignments affecting a CSR's return value. The third form $R' = E$ matches when the other two forms don't match.
	$R' = R''$	Repl. by $R'' \to R$	
	$R' = E$	Repl. by $\top \to R$	
3. Continue the chain backwards from the intermediary towards the function beginning			
$in(S)$ contains $*(SP+c) \to R$	$*(SP+c) = R'$	Repl. by $R' \to *(SP+c) \to R$	We only track stores of registers on the stack. Storing anything else causes all info to be lost.
	$*(SP+c) = E$	Repl. by $\top \to R$	
4. Continue further backwards towards the function beginning			
$in(S)$ contains $R' \to *(SP+c) \to R$	$R' = R''$	Repl. by $R'' \to *(SP+c) \to R$	Again, use of intermediaries other than registers causes all info to be lost.
	$R' = E$	Repl. by $\top \to R$	
5. Adjust stack offsets of restore locations whenever the SP changes			
	$SP = SP+d$	Repl. all $SP+c$ by $SP+c+d$	Adjust stack offset to be relative to current SP. If SP change is complex, approximate to $\top$.
	$SP = E$	Repl. all $SP+c$ by $\top$	
6. Handle merge points			
	`cb` O	$in_1(S) \sqcup in_2(S)$	Merge info from the two successors.
7. In all other cases (including calls and unconditional branches) $out(S) = in(S)$			

Table 5. Equations defining callee-saved register preservation analysis.

that retrieves a saved value from the stack. The rules basically chain the register assignment in the current instruction S with the information captured in $in(S)$ about the restored content of register R. The third subcase of Step 2 handles register assignments that are too complex, e.g., they contain arithmetic operations. In such cases, we conservatively indicate the restored value of R is $\top$.

Step 3 is applicable further up from the point where Step 2 was applied. In particular, it chains known information about the restored content of R with assignments to stack locations. At this point, the restore summary is of the form $R' \to *(SP+c) \to R$. If the value being assigned to the stack location is a complex expression, then we cannot accurately track the source, so we simply summarize it as $\top$.

Step 4 traces back further to determine the source of R', and chains those assignments to update the restore summaries. Once again, if the expression assigned to R' is complex (i.e., it is not another register), then we conservatively indicate that the restored value can be $\top$.

Step 5 handles instructions that modify the stack pointer. If they are of the form that adds or subtracts a constant from the SP, they are easy to handle: we simply update all restore summaries to reflect the value of SP before the increment/decrement operation. For more complex updates to SP, we conservatively indicate the stack location as $\top$.

Step 6 handles merge points, and are further described below. Step 7 captures all other cases of instructions that have no impact on restore summaries, so $out(S)$ is the same as $in(S)$.

4.2 Merge operation

Before defining the merge operator, we add a second component to each restore summary. In particular, $in(S)$ and $out(S)$ become tuples of the form $\langle RS, r\rangle$ where RS stands for a restore summary and r loosely relates to the number of paths in which this summary was found.

As in the case of stack depth analysis, merge operation is based on set union. In particular, tuples from $in_1(S)$ and $in_2(S)$ are compared to identify those with a matching restore summary. They are then combined into one tuple in $out(S)$: $\langle RS, r_1\rangle \sqcup \langle RS, r_2\rangle = \langle RS, r_1 + r_2\rangle$. Tuples from $in_1(S)$ that

don't match any tuple in $in_2(S)$ (or vice-versa) are simply added to $out(S)$. Let s_X be the number of tuples for the callee-saved register X. If $s_X \leq k$, we are done. Otherwise, pick any two tuples $\langle R \rightarrow *(SP+c) \rightarrow X, r_1 \rangle$ and $\langle R \rightarrow *(SP+d) \rightarrow X, r_2 \rangle$ and merge them into $\langle R \rightarrow *(SP+?) \rightarrow X, r_1+r_2 \rangle$. Such a merge is also applied to tuples of the form $\langle *(SP+c) \rightarrow X, r_1 \rangle$ and $\langle *(SP+d) \rightarrow X, r_2 \rangle$. These merge steps are repeated until the number of tuples falls below k, or there are no mergeable tuples. In the latter case, we introduce a second type of merge based on counts. In particular, we remove and set aside tuples with the lowest counts until there are no more than $k-1$ tuples left. These set-aside tuples are merged into a single tuple $\langle ?, r_s \rangle$, where r_s is the sum of the r values being merged. If there is already a tuple with the source "?" we merge the tuples with the lowest counts into this tuple. Note that "?" is evidence of mismatch, and is used in conjunction with the majority rule as proof of non-preservation.

4.3 Supporting BP-relative stack access

The description so far considered the simpler case where all stack accesses are made using SP. However, stack can also be accessing using the base pointer BP. To handle this, we extend restore summaries so that they can either be SP-relative or BP-relative. Specifically, we make a copy of every line of Table 5 that refers to SP, and replace references to SP in the copy with BP. As a result, some restore summaries gathered during analysis may be in terms of SP, while others are in terms of BP. *Note that the restore summary for a register may be BP-relative on some paths and SP-relative on others.* Finally, we add a Step 8 to Table 5 to handle SP to BP assignments and vice-versa.

Step	$in(S)$ contains	S	$out(S)$	Comments
8.1	$\cdots *(SP+d) \rightarrow R$	$SP=BP+c$	$\cdots *(BP+c+d) \rightarrow R$	Here, "$\cdots$" can be empty,
8.2	$\cdots *(BP+d) \rightarrow R$	$BP=SP+c$	$\cdots *(SP+c+d) \rightarrow R$	$\top$ or be some register.

Illustrative Example. We illustrate our analysis on a function that uses a variable amount of stack space, focusing on the register *rbx*. Instructions unaffected by the definition in Table 5 are elided. The explanation column identifies the step from Table 5 that is being applied. When a step n consists of multiple lines, we use the notation $(n.m)$ to refer to the mth line being used. If a line $(n.m)$ is related to the stack, we use the notation $(n.m')$ to refer to the same line but with SP replaced by BP.

Instruction	S	$out(S)$		Explanation
push %rbp	$SP=SP-8$	$BP \rightarrow *(SP-8) \rightarrow BP$	$rbx \rightarrow *(SP-16) \rightarrow rbx$	(5.1)
	$*(SP)=BP$	$BP \rightarrow *(SP) \rightarrow BP$	$rbx \rightarrow *(SP-8) \rightarrow E$	(3.1)
push %rbx	$SP=SP-8$	$*(SP) \rightarrow BP$	$rbx \rightarrow *(SP-8) \rightarrow rbx$	(5.1)
	$*(SP)=rbx$	$*(SP+8) \rightarrow BP$	$rbx \rightarrow *(SP) \rightarrow rbx$	(3.1)
mov %rsp, %rbp	$BP=SP$	$*(SP+8) \rightarrow BP$	$*(SP) \rightarrow rbx$	(8.2)
addq $8, %rbp	$BP=BP+8$	$*(BP+8) \rightarrow BP$	$*(BP) \rightarrow rbx$	(5.1')
…	…			
subq %rcx, %rsp	$SP=SP-rcx$	$*(BP) \rightarrow BP$	$*(BP-8) \rightarrow rbx$	(5.2)
…	…			
movq -8(%rbp), %rbx	$rbx=*(BP-8)$	$*(BP) \rightarrow BP$	$*(BP-8) \rightarrow rbx$	(2.1')
movq %rbp, %rsp	$SP=BP$	$*(BP) \rightarrow BP$	$rbx \rightarrow rbx$	(8.1)
pop %ebp	$BP=*(SP)$	$*(SP) \rightarrow BP$	$rbx \rightarrow rbx$	(2.1)
	$SP=SP+8$	$BP \rightarrow BP$	$rbx \rightarrow rbx$	(5.1)
ret	ret	$BP \rightarrow BP$	$rbx \rightarrow rbx$	(1.1)

5 Implementation

Our implementation of AMACE is written in C++ and builds upon the infrastructure of the DASSA framework [71]. We chose DASSA because it is a modern, public tool that also uses static analysis to support disassembly, providing a strong basis for comparison. Our implementation leverages DASSA's existing components for instruction disassembly and lifting to an intermediate representation (IR).

5.1 Core Backward Analyses

We implemented the three backward analyses techniques described in the previous section:

- *Invalid pointer dereferencing:* This analysis identifies invalid code by propagating the requirement for a value to be initialized (§2). Its output, available for every byte offset l in the text segment, is the list of registers that should be defined at that offset in order to avoid undefined pointer dereferencing when the code starting at l is executed.
- *Stack depth analysis:* This analysis computes stack depth relative to exit points, as described in §3. Specifically, for each offset l, it computes a list of up to k possible values for stack depth at l.
- *Callee-saved register preservation:* This analysis yields two primary outputs: (a) a list of registers actively saved and restored, and (b) a list of restore summaries for the remaining registers. While we implemented the method described in §4, our implementation supports just a single summary per register at this point. As such, the majority voting remains unimplemented for this technique.

5.2 Applications

5.2.1 Function Entry Identification. Existing tools for function entry identification employ sophisticated prioritization techniques to limit their focus to a subset of offsets in the binary. The key benefit of our approach is that, in one shot and in linear time, it can compute all the information that may ever be needed by these tools. Nevertheless, our analysis results need to be used in conjunction with a top-level algorithm that chooses offsets to examine. We therefore use DASSA's initial list of possible and definite function starts as the starting point. (See Sec. 5.1 and 5.2 of [71].) We use our analyses to examine these offsets and report the subset that pass all our static analyses.

5.2.2 Non-Returning Call Identification. We use a three-phase process:

- *Phase 1: Catalog-based identification:* This technique is similar to most existing tools that build a catalog of known non-returning functions.
- *Phase 2: Static analysis of fall-through code:* This phase is similar to that proposed in [71], namely, checking if the fall-through code is a function start. The main difference is that this analysis of follow-through code is invoked after every suspected non-returning call in [71], whereas we simply need to check the results computed in our initial analysis of the superset CFG.
- *Phase 3: Callee analysis:* In this second phase analysis, we use information from the previous phases to expand our list further. In particular, any function all whose paths end in non-returning calls are themselves flagged non-returning, and this information is propagated up the call chain.

Notably, we apply this entire three-phase process to common libraries (e.g., `glibc`) to automatically generate a more comprehensive catalog as compared to the initial list used in Phase 1. This ability reduces the amount of initial manual cataloguing effort needed.

6 Experimental Evaluation

Our experimental evaluation is aimed at answering the following questions:

- **Q1:** How does the accuracy of Amace's byte-level backward analysis stack up against the accuracy of similar forward analyses? We answer this question in §6.1 by comparing with DASSA [71], the only other recent work implementing similar analyses with published software.
- **Q2:** How does the performance of Amace compare with that of previous forward analyses? Specifically, in §6.2, we consider the problem of producing analysis results for every byte offset using each of Amace's static analysis techniques and similar forward analysis.
- **Q3:** Does Amace's implementation achieve the goal of scaling linearly? We answer this question in §6.3 by plotting Amace's runtime as a function of binary size.
- **Q4:** When used for function start identification, how does the accuracy of Amace compare with that of state-of-art binary analysis tools? (§6.4.)

- **Q5:** When used for non-returning call identification, how does the accuracy of Amace compare with that of state-of-art binary analysis tools? (§6.5.)

Dataset. Our evaluation uses Pang et al.'s [61] dataset that has widely been used as the basis to evaluate the accuracy of binary analysis tools [29, 71]. It includes SPEC CPU 2006 binaries and real-world programs (e.g., openssl, nginx, mysqld, and glibc), compiled with GCC and Clang across six optimization levels. For runtime comparisons, we additionally use the StochFuzz dataset [94].

Evaluated Tools. We evaluate Amace against leading binary analysis tools [61], including angr [80], Ghidra [76], Dyninst [35], and Nucleus [3], on the tasks of function starts and non-returning call identification. We also evaluate DASSA [71] to compare the effectiveness of our backward analysis against the equivalent forward analysis on the same function properties. We omit commercial tools due to licensing constraints and learning-based approaches [8] due to reproducibility and bias issues [3].

6.1 Q1: Accuracy comparison with forward analysis

Although the idea of using static analysis to assist in low-level binary code related tasks have been proposed in multiple papers, DASSA [71] is the only recent work for which we were able to find a working implementation of comparable analyses to the ones developed in this paper. In fact, we tailored our static analysis implementation to match their goals so that accuracy comparisons are meaningful.

Accuracy comparisons would ideally be based on ground truths for the information computed by each of our analyses. However, ground truths are unavailable for the properties such as stack depth. Instead, we are evaluating on the ground truth for function starts. The goal of this evaluation is to compare and quantify the relative abilities of DASSA and Amace in predicting function starts. Specifically, we compare on (a) false positives, i.e., the analysis results are consistent with a function start but the offset isn't in the ground truth, and (b) false negatives, i.e., the analysis results are *inconsistent* with a function start but the offset *is* in the ground truth.

False negatives are computed at every function start in the ground truth. For Amace, false positives are evaluated at every byte offset *not* in the ground truth. For DASSA, since it takes too long to evaluate every offset, we used sampling, evaluating FPs at 1000 random offsets *not* in the ground truth.

6.1.1 Invalid pointer dereferencing analysis. Fig. 2 plots the accuracy of Amace's invalid pointer dereferencing analysis against that of DASSA. Note that Amace experiences a modest increase in false negatives from 0.4% to 1.2%. It also achieves a somewhat larger reduction in false positives from 58% to 46%. Although this analysis is not very discriminating, its low false negative rate enables it to be combined with other techniques to reduce false positives with very minimal increase in false negatives.

6.1.2 Stack depth analysis. Fig. 3 plots the accuracy of Amace's backward stack depth analysis against that of DASSA. Note that Amace achieves a significant reduction in false negatives. We believe this is mainly because of the mitigation mechanism we have built for non-returning calls. We expected the false positives to be similar, based on the description of the analysis in the DASSA paper

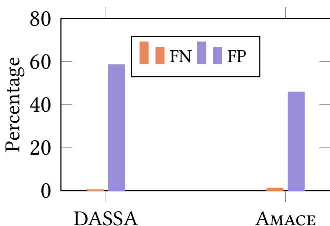


Fig. 2. Invalid pointer analysis

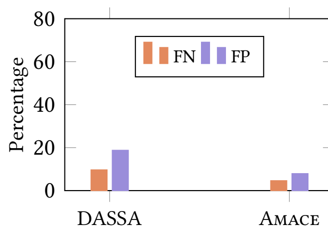


Fig. 3. Stack depth analysis

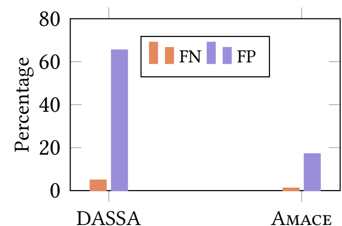


Fig. 4. Register preservation analysis

[71], but our experiments show a somewhat different result. We attribute it to inherent differences between the way forward and backward analyses work, as discussed further below.

6.1.3 Callee-saved register preservation analysis. Fig. 4 compares the accuracy of AMACE's register preservation analysis against DASSA. Note that AMACE produces additional information over DASSA in the form of *actively preserved registers.* This means that the code explicitly saved a register value and then restored it. This factor, and the many differences between forward and backward analyses contribute to a significant increase in accuracy for AMACE. In particular, false negatives decrease from DASSA's 5% to 1% for AMACE, while false positives decrease from 65% to 17%.

6.1.4 Ablation Study. We investigate the contribution of AMACE's components to its overall results.

Stack depth analysis and k. Fig. 5 shows that k (i.e., the maximum size of *out* sets) has only a minimal impact on accuracy. This is due to several factors. First, our experiments show that in 99% of the locations, the set size is exactly one. This is to be expected because functions rarely abuse their stack. Even at unintended instructions, stack depth becomes $\top$, with $k=1$. Second, our implementation does not support $k<3$. If k could be set to 1, then we would likely see a bigger effect. An even more important factor is our non-return call filtering technique, further discussed below.

Register preservation analysis and the number of registers preserved. Fig. 6 plots false negatives and false positives of register preservation analysis if we insist that at least n registers are preserved. Note that on x86/Linux, there are only 6 callee saved registers, so $n=6$ represents a maximum.

There is no such thing as active preservation of zero registers. Thus, $n=0$ is similar to DASSA, which is why we used it for the comparison with DASSA. This setting is characterized by relatively high false positives, which happens because of code paths that don't write into callee-saved registers at all. Setting $n=1$ reduces false positives substantially, but results in an even greater increase to false negatives. This is because many small functions don't use callee saved registers, and because of the active save/restore implied here, they won't pass the criteria for a function. As n increases, false negatives reach almost 100% at $n=6$, with continued decline of false positives.

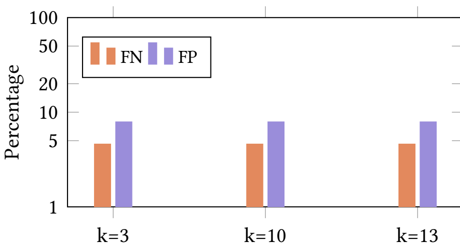


Fig. 5. Stack depth analysis: Accuracy vs k.

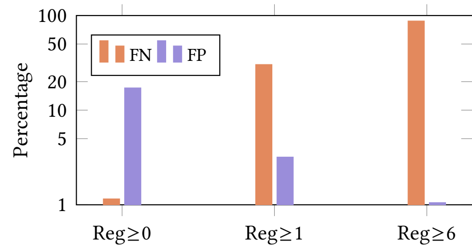


Fig. 6. Accuracy vs. number of registers preserved.

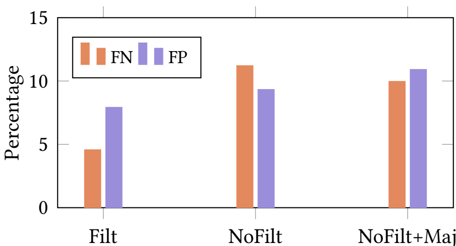


Fig. 7. Effect of non-return filtering and majority voting.

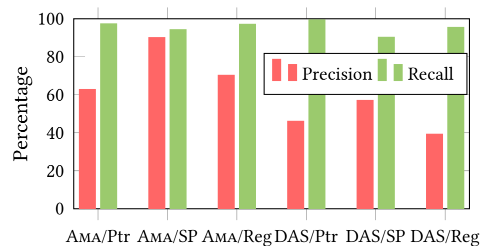


Fig. 8. DASSA vs. AMACE: Function detection accuracy.

Effect of non-returning call filtering and Majority voting. Fig. 7 shows the effect of (a) filtering of non-returning calls, and (b) majority voting. Specifically, "NoFilt" shows false positives and false negatives when Phase 2 of non-return call analysis (§5.2.2) is disabled. Note that it still includes the effect of Phase 1, i.e., filtering out calls to non-returning functions identified by AMACE during its analysis of `libc`.

"NoFilt+Maj" shows the result majority voting feature of stack depth analysis is enabled. It reduces false negatives slightly, but also increases false positives slightly. "Filt" shows the results when Phase 2 of non-return call filtering is applied. Note that it obtains significantly better results than majority voting. The reason is as follows. Instances where Phase 2 recognizes non-returning calls align perfectly with the instances where an inconsistency in stack depths will be observed. However, majority voting is not very reliable because the merge point may occur some distance from the non-returning call. Until such point, bad information about stack depth proceeds unchecked. On top of that, majority voting is necessarily probabilistic, subject to other stack depths reaching merge points. In contrast, Filt promptly eliminates non-returning paths from consideration immediately and deterministically. So, it leads to better results.

6.1.5 Precision and Recall of DASSA vs. AMACE *on Function starts.* Fig. 8 shows the precision and recall achieved by the three analyses discussed above when applied to the function detection problem. The precision and recall numbers reflect the same trends and differences between the two systems shown in Fig.2 to Fig. 4. In particular, recall is given by the simple formula $1-FNR$, where FNR is the false negative rates shown in Fig.2 to Fig. 4. Precision tracks the false positive rates, but its exact value will depend on the prevalence rate of function starts in the input list. Consistent with §6.4, we use DASSA's candidate set as the input.

Fig. 8 shows that all three of AMACE's analyses achieve high recalls, similar to DASSA. Recall rates achieved by stack depth and register preservation analyses are modestly improved over DASSA. On precision, AMACE achieves significant improvement, reflecting the significant reduction in false positives it achieves over DASSA.

6.2 Q2: Runtime comparison with forward analysis

Our goal is to produce static analysis results for every byte offset, so that it can be considered a candidate function start, or as an intended instruction in the binary. We claimed that AMACE is efficient for this problem, whereas previous forward analysis techniques aren't. We show this experimentally in this section. As before, our dataset is from Pang et al. [61].

For forward analysis, repeating the analysis at every offset takes far too long for large binaries, so we rely on a sampling approach. Specifically, we pick m random offsets in a binary of size n, perform a forward analysis starting at the m offsets, and then scale the measured time by n/m to estimate the total runtime for the binary. In our implementation, m is set to 10K. Binaries smaller than 10KB are not sampled this way, but instead we perform forward analysis at every byte offset in the binary.

Figs. 9–11 show the speedup factor achieved by AMACE over DASSA. Leftmost chart shows speedup over each of the three datasets we have used. The speedup ratio is about 700× on the larger datasets, and about half of that on the smaller StochFuzz dataset. The middle and right charts break down the performance by optimization level on the GCC and LLVM datasets. Generally speaking, higher speedups are achieved at higher levels of optimizations.

6.3 Q3: Scalability of AMACE

Fig. 12 plots the runtime of AMACE as a function of binary size. The points are closely clustered around a straight line, demonstrating that we have achieved our goal of linear-time complexity. The

Metric	angr	ghidra	dyninst	nucleus	AMACE
Total runtime (min)	588	1615	13	13	248

Table 6. Total runtime across all binaries.

Metric	angr	ghidra	dyninst	nucleus	AMACE
Peak memory (GiB)	5.49	4.62	2.20	1.13	11.2

Table 7. Peak memory usage across binaries.

fitted straight line is $y = 45.386x + 2.2829$, where x is in MiB and y is in seconds. The fit achieves an R^2 value of 0.965, demonstrating an excellent fit.

Table 6 shows the total runtime across all binaries in minutes, comparing the runtime of AMACE with other contemporary binary analysis tools. It shows that AMACE's runtime falls in the general range of other tools, showing that it is feasible to compute analysis results at every offset in a binary without significantly increasing the runtime of most tools. (A direct comparison of the runtimes of different tools needs to take into account that each tool is targeting a different subset of capabilities, so the times have to be assessed in the context of the capabilities provided by them.)

Fig. 13 plots the peak memory use of AMACE as a function of the binary size. Once again, there is an excellent linear correlation, with an R^2 of 0.984. The fitted model is $y = 1.72x + 0.36$, where x is again in MiB and y is in GiB. Table 7 shows that AMACE's peak memory use is in the general range as many other tools. This shows that the entire superset CFG can be successfully analyzed to produce analysis results at every byte offset without significantly affecting the memory use profile of typical binary analysis tools.

Finally, Fig. 14 shows how the runtime and memory changes with k. As noted earlier, even when we use larger values of k, the size of *out* sets is close to 1 in about 99% of the offsets. As a result, k's value has negligible impact on the runtime and memory use of AMACE.

6.4 Q4: Comparison with State-of-Art Tools: Function starts

Fig. 16 compares the accuracy of AMACE against state-of-the-art tools for binary analysis. Precision is shown as a solid bar, recall as a hollow bar, and the F1-score appears above each bar. For the compared tools, we used the scripts distributed with our dataset — specifically, the artifact for [61] — for our measurements. The scripts compute metrics for each binary in the dataset. They were combined using macro-averaging on the left, and micro-averaging on the right. Note that micro-averaging aggregates true positives, false negatives and false positives across all binaries, thus producing a measure reflecting the fraction of functions identified across the binaries. In contrast, macro-averaging calculates the performance metrics for each binary individually and then computes the arithmetic mean of those scores.

Although AMACE yields a slightly lower macro-averaged F1-score than Nucleus, it outperforms Nucleus under micro-averaging. This demonstrates that despite performing its analysis entirely on the superset CFG, AMACE delivers accuracy comparable to, and in some metrics exceeds, existing tools.

6.5 Q5: Comparison with State-of-Art Tools: Non-returning calls

Fig. 15 compares the effectiveness of AMACE with those of state-of-art tools in binary analysis on the task of identifying non-returning calls. Our results place us in the middle of these tools, once again showing that accurate backward analysis of superset CFG is possible, and can deliver performance comparable to the best tools today. A smaller set of tools address non-returning calls as opposed to function starts, so there are fewer bars in the comparison.

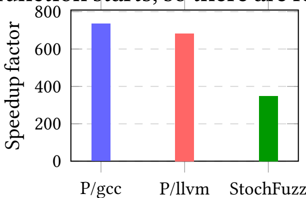


Fig. 9. Speedup across datasets.

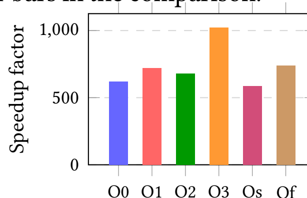


Fig. 10. Speedup vs. optimization level (gcc)

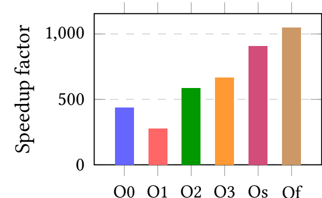


Fig. 11. Speedup vs. optimization level (llvm)

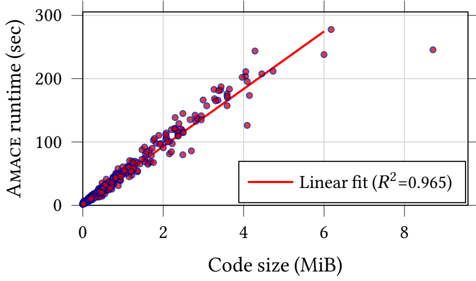


Fig. 12. Runtime vs Code size.

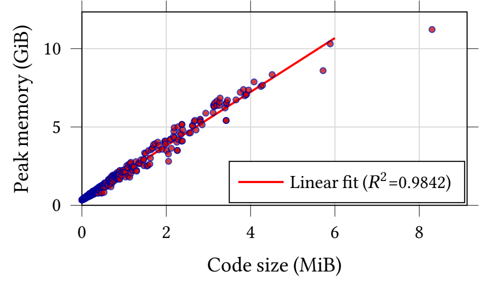


Fig. 13. Peak memory usage vs Code size.

A key point to note is that we obtain our accuracy without relying on catalogs of non-returning functions. Catalogs are the primary source for other tools. In contrast, we obtain these results using the three-phase process in §5.2.2. The ability of our approach to perform this analysis in linear time is crucial in this regard.

7 Related Work

Abstract Interpretation. Static analysis techniques typically rely on abstract interpretation [22] or dataflow analysis [1]. For binaries, *value set analysis* [5] made the important observation of needing to analyze integer values representing memory addresses together with other types of data. This work laid the foundation for addressing the semantic gaps between high-level code and binaries [7] and formed the basis of many binary-based systems and tools [14, 19, 51, 75, 80, 84]. While VSA's domain is well-suited for problems such as the discovery of higher level variables in binaries [6], it is not very effective for reasoning about problems such as register preservation. Saxena et al. proposed a different domain and interpretation [77] targeting this problem. It has been adopted in several works that require an accurate analysis of the effects of functions on registers and the stack [59, 68, 71, 73].

To reconstruct the control-flow structure missing in the binary, systems such as Jakstab [44–46] interleave abstract interpretation with recursive disassembly to iteratively expand the program CFG. Alternatively, abstraction-refinement loops [9] resolve dynamic jumps by tracing data-dependencies

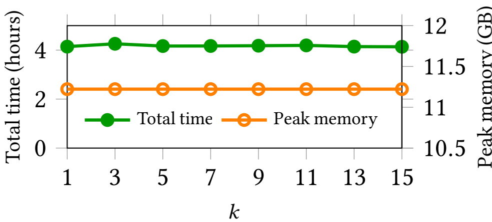


Fig. 14. Runtime and peak memory usage vs. k.

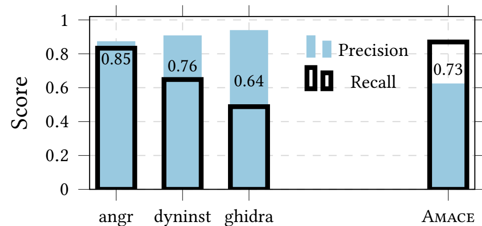
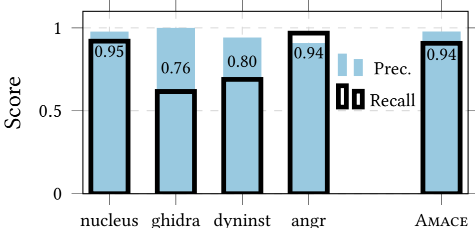


Fig. 15. AMACE vs. SOTA tools: Non-returning call.



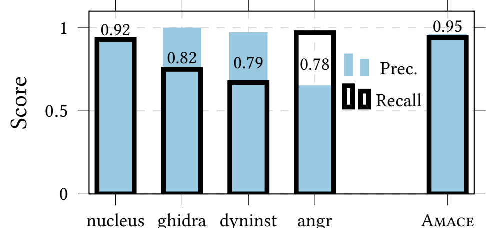


Fig. 16. AMACE vs. SOTA tools: Accuracy of function start identification, macro (left) and micro (right) averages.

backward to locally upgrade precision. While these refinement-based paradigms can incorporate relational analysis to improve precision, they face scalability challenges. Saxena et al.'s stack analysis [77] brings relational capabilities in a scalable setting by focusing on one type of relation: the relation between current register values and the state of registers and the stack at function entry.

Some malware analysis applications [14, 80] require more powerful analyses based on symbolic execution [16–18, 20, 33, 34, 39, 78]. However, these techniques are too heavy-weight to be used when analyzing or instrumenting large code bases.

***DASSA vs.* AMACE.** While the three analyses developed in DASSA and this paper share the same goals, the techniques themselves are very different. Specifically, DASSA uses forward abstract interpretation using Saxena et al.'s domain [77] to track register and stack contents. A generic abstract memory module is used across all three analyses. This memory is initialized at function starts, and updated as each instruction is abstractly executed.

The benefit of using a generic abstract memory module is that it reduces analysis-specific code that needs to be developed for each analysis. Its downside is that it limits what can be specialized for a particular analysis. For instance, this generic treatment can lead to severe accuracy degradation on weak memory updates, i.e., memory stores for which the target cannot be exactly determined. Frequently, this can require the entire stack memory to be marked as "$\top$". DASSA looks for positive evidence of register non-preservation, so this inaccuracy causes it to flag the code as "possible function," thereby significantly increasing false positives. (AMACE side-steps this difficulty since its register preservation does not rely on memory modeling.) DASSA's forward analysis also suffers from higher false negatives because it cannot filter out the effect of non-returning calls in the way AMACE can.

In contrast with DASSA that uses abstract forward execution, AMACE develops custom backward analysis targeting the three analysis problems. Unlike DASSA that can reason about the value of registers and memory, AMACE has to infer constraints that must hold on registers/memory, or relationships between register/memory contents. A good example is register preservation analysis that maintains relationships between registers/stack values at the current instruction and the return instruction. The result is an analysis that requires more development effort but offers better accuracy as well as scalability.

Jump table analysis. Jump table analysis aims to uncover the targets of indirect jumps introduced by the compilation of large switch statements. This is a necessary step before CFGs can be built for functions containing jump tables. Numerous analysis techniques have been developed for this problem [2, 3, 14, 19, 26, 27, 30, 35, 42, 51, 55, 60, 61, 63, 70, 73, 75, 76, 80, 82–84, 86, 89, 91, 92], but most are focused on pragmatic concerns faced by binary tools rather than fundamental improvements in the underlying techniques. By developing a more precise abstract domain targeted for jump table table analysis, and by combining it with improved jump table bounds enabled by a new bidirectional relational analysis, SJA [59] achieves a 3× lower error rates compared to previous works.

One way to cope with inaccuracies in bounds analysis is to validate the targets stored in jump tables. DASSA [71] uses properties such as stack depth for this purpose, i.e., it checks if the stack depth matches between the source and target of the indirect jump. However, it faces scalability challenges because of the need to repeatedly analyze possible targets. (Note that since the CFG hasn't been completed, many of the potential targets may not be intended code.) AMACE avoids this repetition, as it has already computed these properties at every offset using a linear-time analysis of the superset CFG.

Function Entry Identification. Disassembly [30, 71] and control-flow graph construction [27, 35, 55, 59, 76, 80, 80, 86] have critical dependences on function entries. Metadata, in the form of debug or exception-handling info [2, 30, 62, 69, 71, 76, 86], is often the most reliable source of function starts, but it may be absent or incomplete in general. Function prolog patterns are the most commonly used

metadata-independent technique [14, 35, 50, 55, 57, 76, 80], but they are prone to false positives as well as negatives. Another common heuristic is to treat locations following previously identified functions as new function starts, often assisted by additional assumptions such as the ability of linear disassembly to discover all code [3, 28, 70, 86]. These heuristics can be fooled by data or padding embedded in code. Finally, the static analyses described in this paper have been used in [71, 73].

All of the above-mentioned techniques suffer from significant error rates on their own. Achieving even a 90% accuracy typically requires an elaborate, fine-tuned orchestration of these techniques, such as the prioritized error correction algorithm of DASSA. Since our goal in this paper has been more limited — showing that Amace can achieve accuracies comparable to forward analyses — we have not undertaken such an effort yet; and instead relied on a candidate set of possible function starts computed by DASSA before this phase. We believe that a carefully engineered combination can achieve significant improvements in accuracy over that reported in this paper.

Machine-learning approaches [8, 64, 79, 88] can be a more systematic way to identify patterns associated with function starts, but correlations between training and test data are hard to avoid in this domain [3], leading to overfitting. Consequently, current binary tools [3, 35, 76, 80] continue to rely on conventional function detection techniques.

Non-Returning Call Identification. Non-returning calls, such as `exit` or `abort`, pose a major challenge for accurate disassembly [61]. Many existing tools rely on carefully curated catalogs of known non-returning functions in system libraries [14, 35, 76, 80], possibly augmented with static analysis to discover additional functions that *always* call non-returning functions [14, 55, 61]. However, this doesn't account for non-returning indirect calls and non-returning calls to functions that *sometimes* don't return. To cover these cases and to reduce dependence on catalog completeness, DASSA flags a non-returning call if the following offset satisfies the criteria for a function start. However, this leads to the familiar scalability challenge that is solved by Amace.

Amace's ability to detect non-returning calls — as opposed to other tools that detect *calls* to non-returning functions — is accurate enough that it doesn't have to rely on an initial catalog. Nevertheless, our current approach detects only a subset of non-returning calls, namely, those followed by another function. Other cases are missed, e.g., a call followed by an intra-function jump target where the stack depth doesn't match that of the call. To detect these cases, we need better techniques to detect and resolve inconsistencies in stack depths at merge points above non-returning calls.

8 Conclusions

Although numerous frameworks and methods exist for static binary analysis, all of them operate after an initial disassembly. To our knowledge, ours is the first work to develop sophisticated static analysis that can operate before committing to a disassembly. We began with our key insight that if we focus on backward analyses, we can ensure accuracy of analysis results at intended instructions even though we have no idea where these intended instructions are. Moreover, our analysis operates in time that is linear in the size of the binary. For the task it is intended for, namely, providing static analysis results for any offset in a binary, it is orders of magnitude faster than previous techniques. Finally, our evaluation results also show that these performance benefits are achieved without sacrificing analysis accuracy.

9 Data-Availability Statement

The Amace source code and relevant scripts are available at https://github.com/nhuhuan/amace. These resources, including the artifact [58], are intended to enable the verification of our results, and support further research into scalable static analysis techniques capable of analyzing raw bytes.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. 1985. *Compilers: Principles, Techniques and Tools.* Addison-Wesley.

[2] Jim Alves-Foss and Jia Song. 2019. Function boundary detection in stripped binaries. In *ACSAC*. doi:10.1145/3359789.3359825

[3] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-agnostic function detection in binaries. In *IEEE S&P*. doi:10.1109/EuroSP.2017.11

[4] Michael Backes and Stefan Nürnberger. 2014. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *USENIX Security*.

[5] G. Balakrishnan and T. Reps. 2004. Analyzing memory accesses in x86 executables. In *Compiler Construction*. doi:10.1007/978-3-540-24723-4_2

[6] G. Balakrishnan and T. Reps. 2007. Divine: Discovering variables in executables. In *VMCAI*.

[7] Gogul Balakrishnan and Thomas Reps. 2010. WYSINWYX: What you see is not what you eXecute. *ACM Trans. Program. Lang. Syst.* (2010).

[8] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. 2014. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *USENIX Security*.

[9] Sebastien Bardin, Philippe Herrmann, and Franck Vedrine. 2011. Refinement-based CFG reconstruction from unstructured programs. In *VMCAI*.

[10] Erick Bauman, Zhiqiang Lin, and Kevin W Hamlen. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics.. In *NDSS*.

[11] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. 2005. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security*.

[12] Edson Borin, Cheng Wang, Youfeng Wu, and Guido Araujo. 2006. Software-Based Transparent and Comprehensive Control-Flow Error Detection. In *CGO*.

[13] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *CGO*.

[14] D. Brumley, I. Jager, T. Avgerinos, and E. Schwartz. 2011. Bap: a binary analysis platform. In *CAV*. doi:10.1007/978-3-642-22110-1_37

[15] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. 2006. Detecting self-mutating malware using control-flow graph matching. In *DIMVA*.

[16] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *OSDI*.

[17] Cristian Cadar and Dawson Engler. 2005. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *Proceedings of the 12th International Conference on Model Checking Software* (San Francisco, CA) *(SPIN'05)*. 2–23.

[18] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM conference on Computer and communications security* (Alexandria, Virginia, USA) *(CCS '06)*. 322–335.

[19] Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang. 2021. SelectiveTaint: Efficient Data Flow Tracking With Static Binary Rewriting. In *USENIX Security*.

[20] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. 2009. Selective symbolic execution. In *Workshop on Hot Topics in Dependable Systems*.

[21] Cristina Cifuentes and Mike Van Emmerik. 2001. Recovery of jump table case statements from binary code. *Science of Computer Programming* 40, 2-3 (2001), 171–188.

[22] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages (POPL)*.

[23] Marco Cova, Viktoria Felmetsger, Greg Banks, and Giovanni Vigna. 2006. Static detection of vulnerabilities in x86 executables. In *ACSAC*.

[24] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. 2015. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *NDSS*.

[25] Lucas Vincenzo Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. 2013. Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *ACM CCS*.

[26] Alessandro Di Federico and Giovanni Agosta. 2016. A jump-target identification method for multi-architecture static binary translation. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. doi:10.1145/2968455.2968514

[27] 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*. doi:10.1145/3033019.3033028

[28] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *IEEE S&P*.

[29] Antonio Flores-Montoya, Junghee Lim, Adam Seitz, Akshay Sood, Edward Raff, and James Holt. 2025. Disassembly as Weighted Interval Scheduling with Learned Weights. In *IEEE S&P*.

[30] Antonio Flores-Montoya and Eric Schulte. 2020. Datalog disassembly. In *USENIX Security*.

[31] Lian Gao and Heng Yin. 2025. BinDSA: Efficient, Precise Binary-Level Pointer Analysis with Context-Sensitive Heap Reconstruction. *ISSTA* (2025).

[32] Masoud Ghaffarinia and Kevin W Hamlen. 2019. Binary control-flow trimming. In *ACM CCS*.

[33] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *PLDI*.

[34] Patrice Godefroid, Michael Y Levin, and David A Molnar. 2008. Automated Whitebox Fuzz Testing. In *Network Distributed Security Symposium (NDSS)*, Vol. 8. 151–166.

[35] Laune C Harris and Barton P Miller. 2005. Practical analysis of stripped binary code. *ACM SIGARCH* (2005). doi:10.1145/1127577.1127590

[36] Niranjan Hasabnis, Rui Qiao, and R Sekar. 2015. Checking correctness of code generator architecture specifications. In *CGO*.

[37] Niranjan Hasabnis and R Sekar. 2016. Extracting Instruction Semantics Via Symbolic Execution of Code Generators. In *ACM FSE*.

[38] Niranjan Hasabnis and R Sekar. 2016. Lifting assembly to intermediate representation: A novel approach leveraging compilers. In *ASPLOS*. doi:10.1145/2872362.2872380

[39] Klaus Havelund and Thomas Pressburger. 2000. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer* (2000).

[40] Xin Hu and Kang G Shin. 2013. DUET: integration of dynamic and static analyses for malware clustering with cluster ensembles. In *ACSAC*.

[41] IDApro [n. d.]. Hex rays. https://www.hex-rays.com/index.shtml.

[42] Sun Hyoung Kim, Cong Sun, Dongrui Zeng, and Gang Tan. 2021. Refining Indirect Call Targets at the Binary Level. In *NDSS*. doi:10.14722/ndss.2021.24386

[43] Sun Hyoung Kim, Dongrui Zeng, Cong Sun, and Gang Tan. 2022. BinPointer: towards precise, sound, and scalable binary-level pointer analysis. In *ACM SIGPLAN International Conference on Compiler Construction*.

[44] Johannes Kinder. 2010. *Static analysis of x86 executables*. Technical Report. Technische Universität Darmstadt.

[45] Johannes Kinder and Helmut Veith. 2008. Jakstab: A Static Analysis Platform for Binaries. In *CAV*.

[46] Johannes Kinder, Florian Zuleger, and Helmut Veith. 2009. An Abstract Interpretation-Based Framework for Control Flow Reconstruction From Binaries. In *VMCAI*.

[47] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. 2002. Secure Execution via Program Shepherding. In *USENIX Security*.

[48] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P Kemerlis, and Michalis Polychronakis. 2018. Compiler-assisted code randomization. In *IEEE S&P*. doi:10.1109/SP.2018.00029

[49] Hyungjoon Koo and Michalis Polychronakis. 2016. Juggling the gadgets: Binary-level code randomization using instruction displacement. In *Asia CCS*.

[50] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. 2004. Static disassembly of obfuscated binaries. In *USENIX Security*.

[51] Draper Laboratory. 2022. *CBAT: A Comparative Binary Analysis Tool*. Technical Report. ONR TPCP.

[52] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *NDSS* (2018).

[53] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. 2014. Automatic runtime error repair and containment via recovery shepherding. *PLDI* (2014). doi:10.1145/2666356.2594337

[54] C-K Luk, Robert Muth, Harish Patil, Robert Cohn, and Geoff Lowney. 2004. Ispike: a post-link optimizer for the intel/spl reg/itanium/spl reg/architecture. In *CGO*.

[55] Xiaozhu Meng and Barton P Miller. 2016. Binary code is not easy. In *ISSTA*. doi:10.1145/2931037.2931047

[56] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. 2019. Probabilistic disassembly. In *IEEE/ACM ICSE*.

[57] Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi-cker Chiueh. 2006. BIRD: Binary interpretation using runtime disassembly. In *CGO*.

[58] Huan Nguyen. [n. d.]. Analyzing Bytes: Pre-Disassembly Static Binary Analysis. doi:10.5281/zenodo.19688632

[59] Huan Nguyen, Soumyakant Priyadarshan, and R Sekar. 2024. Scalable, Sound, and Accurate Jump Table Analysis. In *ISSTA*.

[60] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. Bolt: a practical binary optimizer for data centers and beyond. In *IEEE/ACM CGO*. doi:10.1109/CGO.2019.8661201

[61] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. 2021. SoK: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *IEEE S&P*. doi:10.1109/SP40001.2021.00012

[62] Chengbin Pang, Ruotong Yu, Dongpeng Xu, Eric Koskinen, Georgios Portokalidis, and Jun Xu. 2021. Towards Optimal Use of Exception Handling Information for Function Detection. In *DSN*.

[63] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. 2012. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE S&P*. doi:10.1109/SP.2012.41

[64] Kexin Pei, Jonas Guan, David Williams-King, Junfeng Yang, and Suman Jana. 2021. Xda: Accurate, robust disassembly with transfer learning. *NDSS* (2021).

[65] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *IEEE S&P*.

[66] Pin [n. d.]. Pin - A Dynamic Binary Instrumentation Tool. http://pintool.org/.

[67] Aravind Prakash and Heng Yin. 2015. Defeating rop through denial of stack pivot. In *Proceedings of the 31st Annual Computer Security Applications Conference*. 111–120.

[68] Soumyakant Priyadarshan, Huan Nguyen, Rohit Chouhan, and R Sekar. 2023. SAFER: Efficient and Error-Tolerant Binary Instrumentation. In *USENIX Security*.

[69] Soumyakant Priyadarshan, Huan Nguyen, and R. Sekar. 2020. On the Impact of Exception Handling Compatibility on Binary Instrumentation. In *ACM FEAST*.

[70] Soumyakant Priyadarshan, Huan Nguyen, and R. Sekar. 2020. Practical Fine-Grained Binary Code Randomization. In *ACSAC*. doi:10.1145/3427228.3427292

[71] Soumyakant Priyadarshan, Huan Nguyen, and R. Sekar. 2023. Accurate Disassembly of Complex BinariesWithout Use of Compiler Metadata. In *ASPLOS*.

[72] Chenxiong Qian, Hong Hu, Mansour Alharthi, Simon Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating.. In *USENIX Security*.

[73] Rui Qiao and R Sekar. 2017. A Principled Approach for Function Recognition in COTS Binaries. In *Dependable Systems and Networks (DSN)*. doi:10.1109/DSN.2017.29

[74] Rui Qiao, Mingwei Zhang, and R Sekar. 2015. A Principled Approach for ROP Defense. In *ACSAC*.

[75] Nilo Redini, Ruoyu Wang, Aravind Machiry, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. 2019. BinTrimmer: Towards static binary debloating through abstract interpretation. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 482–501.

[76] Roman Rohleder. 2019. Hands-on Ghidra - A Tutorial about the Software Reverse Engineering Framework. In *Proceedings of the 3rd ACM Workshop on Software Protection*. doi:10.1145/3338503.3357725

[77] Prateek Saxena, R Sekar, and Varun Puranik. 2008. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *CGO*.

[78] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *FSE*.

[79] Eui Chul R ichard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing functions in binaries with neural networks. In *USENIX Security*.

[80] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *IEEE S&P*. doi:10.1109/SP.2016.17

[81] 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 *WCRE*.

[82] Victor Van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical context-sensitive CFI. In *ACM CCS*. doi:10.1145/2810103.2813673

[83] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *IEEE S&P*. doi:10.1109/SP.2016.60

[84] 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*. doi:10.14722/ndss.2017.23225

[85] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. 2012. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *ACM CCS*.

[86] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *ASPLOS*. doi:10.1145/3373376.3378470

[87] Miuyin Yong Wong, Matthew Landen, Manos Antonakakis, Douglas M Blough, Elissa M Redmiles, and Mustaque Ahamad. 2021. An inside look into the practice of malware analysis. In *ACM CCS*. doi:10.1145/3460120.3484759

[88] Sheng Yu, Yu Qu, Xunchao Hu, and Heng Yin. 2022. DeepDi: Learning a relational graph convolutional network model on instructions for fast and accurate disassembly. In *USENIX Security*.
[89] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *IEEE S&P*. doi:10.1109/SP.2013.44
[90] Mingwei Zhang, Michalis Polychronakis, and R Sekar. 2017. Protecting COTS Binaries from Disclosure-guided Code Reuse Attacks. In *ACSAC*.
[91] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R Sekar. 2014. A platform for secure static binary instrumentation. *ACM VEE* (2014). doi:10.1145/2576195.2576208
[92] Mingwei Zhang and R Sekar. 2013. Control flow integrity for COTS binaries. In *USENIX Security*.
[93] Mingwei Zhang and R Sekar. 2015. Control flow and code integrity for COTS binaries: An effective defense against real-world ROP attacks. In *ACSAC*.
[94] Zhuo Zhang, Wei You, Guanhong Tao, Yousra Aafer, Xuwei Liu, and Xiangyu Zhang. 2021. Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting. In *IEEE S&P*.

Received 2025-11-14; accepted 2026-04-03