Precise Calling Context Encoding

William N. Sumner, Yunhui Zheng, Dasarath Weeratunge, and Xiangyu Zhang, *Member*, *IEEE*

Abstract—Calling contexts (CCs) are very important for a wide range of applications such as profiling, debugging, and event logging. Most applications perform expensive stack walking to recover contexts. The resulting contexts are often explicitly represented as a sequence of call sites and hence are bulky. We propose a technique to encode the current calling context of any point during an execution. In particular, an acyclic call path is encoded into one number through only integer additions. Recursive call paths are divided into acyclic subsequences and encoded independently. We leverage stack depth in a safe way to optimize encoding: If a calling context can be safely and uniquely identified by its stack depth, we do not perform encoding. We propose an algorithm to seamlessly fuse encoding and stack depth-based identification. The algorithm is safe because different contexts are guaranteed to have different IDs. It also ensures contexts can be faithfully decoded. Our experiments show that our technique incurs negligible overhead (0-6.4 percent). For most medium-sized programs, it can encode all contexts with just one number. For large programs, we are able to encode most calling contexts to a few numbers. We also present our experience of applying context encoding to debugging crash-based failures.

Index Terms—Calling context, context sensitivity, profiling, path encoding, calling context encoding, call graph

1 Introduction

The goal of calling context (CC) encoding is to uniquely represent the current context of any execution point using a small number of integer identifiers (IDs), ideally just one. Such IDs are supposed to be automatically generated at runtime by program instrumentation. Efficient context encoding is important for a wide range of applications.

Event logging is essential to understanding runtime interactions between different components of large distributed or parallel systems. However, different modules in these systems tend to use the same library to communicate, e.g., sending a message using a socket library. Simply logging these communication events often fails to record the intents of these events. Recording their contexts would be very informative. On the other hand, it would be expensive and bulky, as it often implies walking stack frames to reconstruct a context and explicitly dumping the context as a sequence of symbolic function names. It has been shown in [1] that context sensitive event logging is critical for event reduction, which speeds up execution replay by removing events in a replay log that are not relevant to producing a failure. In that work, context information was retrieved through expensive stack walks. Calling contexts have also been used to reverse engineer the format of program input in AUTOFORMAT [2]. In aspect-oriented programming, properties of calling contexts may be used to precisely locate aspects and have been used to support gathering execution information for debugging and unit test generation [3]. Context information has been shown to be very useful in testing sensor network applications in [4].

• *The authors are with the Department of Computer Science, Purdue University, 305 North University Street, West Lafayette, IN 47907-2107. E-mail: {wsumner, zheng16, dweeratu, xyzhang}@cs.purdue.edu.*

Manuscript received 6 July 2010; revised 22 Dec. 2010; accepted 7 May 2011; published online 25 July 2011.
Recommended for acceptance by M. Jackson.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2010-07-0211.
Digital Object Identifier no. 10.1109/TSE.2011.70.

Context encoding can also improve bug reporting. The backtrace of a failure, itself a context, is a very useful component in a bug report. With context encoding, succinct bug reports can be generated. Moreover, it is also possible to collect contexts of additional execution points besides the point of failure. For programs without symbolic information (for the sake of intellectual property protection), context encoding provides a way to anonymously represent contexts and allows them to be decoded at the developers' site.

Context sensitive profiling is very important to program optimization [5], [6], [7]. It annotates program profiles, such as execution frequencies, dependences, and object lifetimes, with context information. Stack walking is too expensive when profile information is generated at a high frequency. Context sensitive optimizations [8], [9] often specify how programs should behave in various contexts to achieve efficiency. For example, region-based memory management [8] tries to cluster memory allocations into large chunks, called regions, so that they can be explicitly managed; context sensitive region-based memory management specifies in which region an allocation should be performed under various contexts. Such analyses need to disambiguate the different contexts reaching a program point at runtime to decide if the current context is one of those specified. Context encoding is highly desirable in this case.

Realizing the importance of context encoding, in recent years a few encoding methods have been proposed. In [6] and [10], a technique is proposed to instrument call sites to cumulatively compute a hash of the function and line number containing the call site. The same encoding is guaranteed to be produced if the same context is encountered because the hash function is applied over the same data in the same order. The technique lacks a robust decoding capability, meaning the context cannot necessarily be decoded from the computed hash. Note that such capability is essential to applications that require inspecting and understanding contexts. Moreover, different contexts may have the same encoding.

In [11], an approach is proposed to change stack frame sizes by allocating extra space on stack frames such that the stack offset, which is essentially the aggregation of stack frame sizes, disambiguates the contexts. This approach is not safe either, especially in the presence of recursion. The reason is that the number of stack frames at runtime could be arbitrary such that the aggregated size cannot be statically analyzed or safely profiled. Hence, the extra space on individual stack frames cannot be safely determined. The technique relies on offline training to generate a decoding dictionary. Both the inherent imprecision and incompleteness in the training set may lead to failures during decoding. The reported failure rate could be as high as 27 percent [11].

In this paper, we leverage the Ball-Larus (BL) control flow encoding algorithm to solve the context encoding problem. Context encoding has different constraints such that a more efficient algorithm can be devised. The basic idea is to instrument function calls with additions to an integer ID such that the value of the ID uniquely identifies contexts. Our algorithm is safe, uniquely identifying different contexts. It can precisely recover a context from its encoding. It has low overhead and handles function pointers, stack allocations, recursion, and so on.

Our main contributions are summarized as follows:

- We leverage the BL algorithm to encode acyclic contexts. The algorithm is more efficient as it exploits the unique characteristics of calling context encoding.
- We propose an algorithm to encode recursive contexts. We divide recursive contexts into acyclic subsequences that are then encoded independently. The subsequence encodings are stored to a stack. A featherweight generic compression further reduces the stack depth.
- We propose an algorithm to safely leverage stack depths to disambiguate contexts. Instead of allocating extra space on the stack to distinguish contexts, we use our acyclic encoding algorithm lazily, meaning that we apply it only to contexts that cannot be safely disambiguated through stack offsets.
- In the presence of recursive contexts, we may have to use a sequence of IDs to encode a context with each ID representing an acyclic subsequence. We propose an algorithm that can effectively unroll the recursion to allow encoding with fewer IDs. The novelty of the algorithm is that it does not unroll the program, which may lead to undesirable code explosion. Instead, the unrolling is abstract, done merely for computing the proper instrumentation for the original program. Informally, it groups sequences of recursive calls into single-path IDs.
- We have a publicly available prototype implementation [12]. We evaluate it on a set of SPEC benchmarks and other large real-world benchmarks. Our experiments show that our technique has very low overhead (0-6.4 percent) and it can encode all contexts of most medium-sized programs with just one 32-bit integer. For large programs, it can encode most runtime contexts with a few numbers. The proposed unrolling algorithm can substantially reduce the maximum number of IDs needed to encode a recursive context.
- We also study context encoding with a particular use case. We propose a debugging technique that is suitable for crash-based failures. The technique records the latest definition point of each variable and the context of the definition. Hence, when a crash occurs due to memory corruption, we can trace back to the instruction that corrupted the memory and, more importantly, the context under which the corruption occurred, because the corruption point itself is often not buggy. For example, it may often be in library code. The context of the corruption point reveals a wealth of information about the root cause. Encoding allows us to efficiently represent and store the context for each memory write. Our results show that the technique is effective.

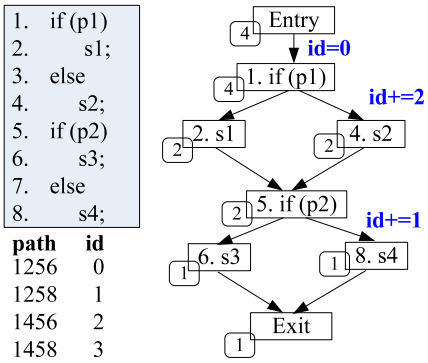


Fig. 1. Example for Ball-Larus path encoding. Instrumentation is marked on control flow edges. Node annotations (rounded boxes at the corners) represent the number of paths starting from the annotated point.

The organization of the paper is as follows: Section 2 motivates our technique by showing the inappropriateness of the BL control flow encoding algorithm. Section 3 presents the basic definitions used in the paper. Section 4 introduces the basic encoding algorithm for handling acyclic calling contexts. Section 5 describes how to encode recursive contexts. In Section 6, we introduce the hybrid encoding algorithm that safely leverages stack depths to disambiguate contexts and avoid explicit encoding. We describe our unrolling algorithm in Section 8. Section 7 presents how we handle various practical issues. Evaluation results are presented in Section 9. Related work is discussed Section 11 and conclusions are given in Section 12.

2 Motivation

2.1 Background: Ball-Larus Path Encoding

In the seminal paper [13], Ball and Larus proposed an efficient algorithm (referred to as the BL algorithm) to encode intraprocedural control flow paths taken during execution. The basic BL algorithm translates an acyclic path encoding into instrumentation on control flow edges. At runtime, a sequence of instrumentation is executed following a control flow path, resulting in the path identifier being computed. All instrumentation involves only simple additions. The idea can be illustrated by the example in Fig. 1. Code is shown on the left and its control flow graph is

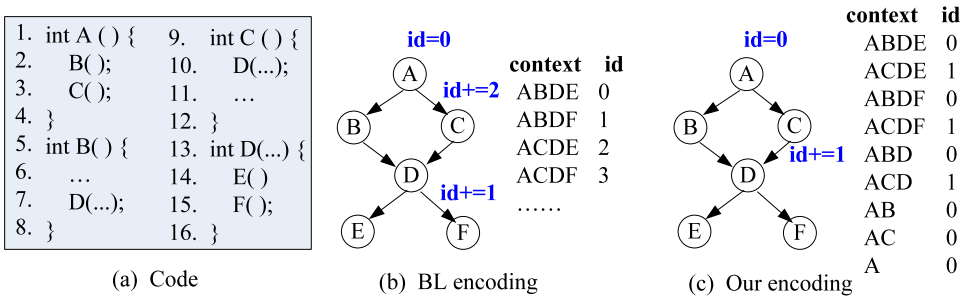


Fig. 2. The inappropriateness of the BL algorithm for encoding contexts.

shown on the right. Instrumentation is marked on control flow edges. Before the first statement, *id* is initialized to 0. If the false branch is taken at line 1, *id* is incremented by 2. If the false branch is taken at line 5, *id* is incremented by 1. As shown in the bottom left, executions taking different paths lead to different values in *id*. Simply, *id* encodes the path.

The algorithm first computes the number of paths leading from each node to the end of the procedure. For example, node 1 has four paths to `Exit`. Such numbers are shown as rounded boxes in Fig. 1. Given a node with n paths, the instrumentation from the node to `Exit` generates IDs falling into the range of $[0, n)$ to denote the paths. For instance, the instrumentation in Fig. 1 generates paths with IDs in [0,4). Namely, the instrumentation on $1 \rightarrow 4$ separates the range into two subranges: $[0, 2)$ and $[2, 4)$, denoting the paths following edges $1 \rightarrow 2$ and $1 \rightarrow 4$, respectively. The instrumentation on $5 \rightarrow 8$ further separates the two paths from 5 to `Exit`.

More details about the BL algorithm can be found in [13]. The algorithm has become canonical in control flow encoding and been widely used in many applications [14], [15], [16].

2.2 Inappropriateness of BL for Context Encoding

Although the BL algorithm is very efficient at encoding control flow paths, we observe that it is inadequate for encoding calling contexts, which are essentially paths in a call graph (CG). Consider the example in Fig. 2. Code is shown on the left. Figs. 2b and 2c show its call graph with two encoding schemes. The BL encoding is presented in Figs. 2b, and 2c shows a more efficient encoding. Nodes in a call graph represent functions and edges represent function calls. BL encoding is designed to encode statement granularity paths leading from the entry of a function to the end of it. The criterion is that each of these paths has a unique encoding. As shown in Fig. 2b, all the paths leading from `A` to `E` or `F` have different encodings. However, context encoding has a different criterion, that is, *all unique paths leading from the root to a specific node have unique encodings,= because we only need to distinguish the different contexts with respect to that node.* In other words, there are no constraints between the encodings of paths that end at different nodes. It is fine if two paths that end at different nodes have the same encoding. For example, paths `ABDE` and `ABDF` have the same encoding 0 according to the scheme in Fig. 2c. As a result, although the encoding in Fig. 2c has less instrumentation (on 2 edges versus 3 in Fig. 2b) and requires a smaller encoding space (the maximum ID is 1 versus 3 in Fig. 2b), it still clearly distinguishes the various contexts of any node.

Our technique is based on the above observation. It handles recursive calls and function pointers. It also leverages the stack offset, namely, the offset of the stack pointer regarding the stack base, to achieve efficiency. More importantly, our algorithm is precise, meaning it ensures that each context has a unique encoding and it allows decoding.

3 Definitions

Definition 1. *A call graph is a pair $\langle N, E\rangle$. N is a set of nodes with each node representing a function. E is a set of directed edges. Each edge $e \in E$ is a triple $\langle n, m, \ell\rangle$, in which $n, m \in N$ represent a caller and callee, respectively, and ℓ represents a call site where n calls m.*

In the above definition of call graph, call edges are modeled as a triple instead of a just the caller and callee pair because we want to model the cases in which the caller may have multiple invocations to the callee.

Definition 2. *The calling context of a given function invocation m is a path in the CG leading from the root node to the node representing m.*

The context of an execution point can be computed by concatenating the context of its enclosing function invocation and the program counter (PC) of the point.

Definition 3. *A valid calling context encoding scheme is a function $En : CC \rightarrow \overline{Z}$ such that*

$$\forall n \in N, \forall x, y \in \{the\ \text{CCs of}\ n\} \wedge x \neq y, En(x) \neq En(y),$$

where CC denotes the set of all possible calling contexts.

Any encoding scheme that generates unique encodings, i.e., integer sequences ($\overline{\mathbb{Z}}$ represents a sequence of integers), for unique contexts of the same function is a valid encoding scheme. For example, a naive but valid encoding scheme is to use the sequence of call site PCs to denote a context.

Our research challenge is to devise a highly efficient valid encoding scheme which also allows precise decoding. Fig. 2c presents an example of such a scheme.

4 Encoding Acyclic Graphs

In this section, we introduce an algorithm to encode calling contexts that do not involve recursion. The basic idea is illustrated in Fig. 3. Assume function $numCC(n)$ represents the number of contexts of a node n such that $numCC(n) = \sum_{i=1...m} numCC(p_i)$, where the predecessors of n are p_i for

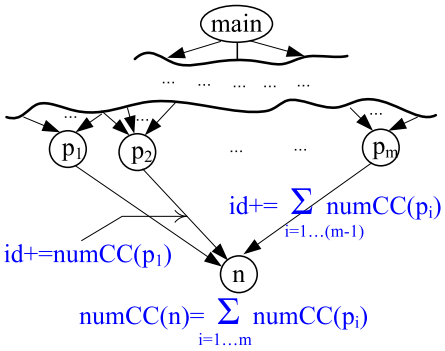


Fig. 3. Intuition of our algorithm.

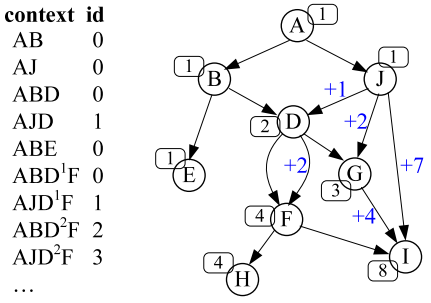


Fig. 4. Example for acyclic encoding. An edge label "$+c$" means "$id+=c$" is added before the invocation and "$id-=c$" is added after; the superscript on D is to disambiguate call sites.

$i \in [1, m]$. A critical invariant of our technique is that *the $numCC(n)$ contexts of n should be encoded by the numbers in the range of $[0, numCC(n))$*. To do so, the edge instrumentation should separate the range into m disjoint subranges, with $[0, numCC(p_1))$ representing the $numCC(p_1)$ contexts along edge $p_1 \to n$, $[numCC(p_1), numCC(p_1) + numCC(p_2))$ representing the $numCC(p_2)$ contexts along $p_2 \to n$, and $[\sum_{j=1\ldots(i-1)} numCC(p_j), \sum_{j=1\ldots i} numCC(p_j))$ encoding the $numCC(p_i)$ paths along $p_i \to n$. As shown in Fig. 3, this can be achieved by instrumenting an edge $p_i \to n$ with "$id+= \sum_{j=1\ldots(i-1)} numCC(p_j)$."

The algorithm is presented in Algorithm 1. It first computes the number of contexts for each node (stored in $numCC$). It then traverses each node n in the main loop in lines 8-14. For each edge $e = \langle p, n, \ell \rangle$, the following instrumentation is added: Before the invocation at ℓ, the context identifier id is incremented by the sum s of the $numCC$s of all preceding callers; after the invocation, id is decremented by the same amount to restore its original value.

Algorithm 1. Encoding for Acyclic CGs

```
1: Annotate (N, E) {
2:     for n ∈ N in topological order do:
3:         for each predecessor p of n do:
4:             numCC[n] ← numCC[n] + numCC[p]
5: }
6: Instrument (N, E) {
7:     Annotate (N,E)
8:     for n ∈ N do:
9:         s ← 0
10:        for each e = ⟨p, n, ℓ⟩ in E do:
11:            annotate e with "+s"
12:            insert [id = id + s] before ℓ
13:            insert [id = id − s] after ℓ
14:            s ← s + numCC[p]
15: }
```

Consider the example in Fig. 4. Node annotations (i.e., numbers in boxes) are first computed. In a topological traversal, nodes A, B, and J are annotated with 1, meaning these functions have only one context; node D's annotation is the sum of those of B and J, denoting there are two possible contexts when D is called; the remaining nodes are similarly annotated. The program is instrumented based on the annotations. Consider the invocations to I, which are from F, G, and J. The empty label on edge FI means that the invocation is instrumented with "$id+=0$" and "$id-=0$" before and after the call, respectively. This instrumentation is optimized away. The edge GI has the label "+4," meaning the instrumentation before and after comprises "$id+=4$" and "$id-=4$." Note that 4 is the annotation of F. Similarly, since the sum of the annotations of F and G is $4+3=7$, edge JI has the label "+7." Other edges are similarly instrumented. At runtime, the instrumentation yields the encodings as shown in the table in Fig. 4.

In applications such as context sensitive profiling and context sensitive logging, context IDs are emitted as part of the profile. In order to facilitate human inspection, decoding is often needed. The decoding algorithm is presented in Algorithm 2. The algorithm traverses from the given function in a bottom-up fashion and recovers the context by comparing the encoding with edge annotations. In particular, at line 2, the recovered context cc is initialized with the given function m, which is the leaf node of the final context. Lines 4-11 are the main process, which terminates when the root node is reached. In the inner loop from lines 5 to 9, the algorithm traverses edges ending at the current function n. At line 6, it tests if the encoding falls in the encoding range of the contexts along the current edge. If so, the edge is taken; the caller p and the call site ℓ are attached to the recovered context at line 7. Symbol "$\bullet$" represents concatenation. At line 8, the encoding is updated by subtracting the edge annotation. This essentially reverses one step of encoding. The process continues by treating the caller as the new current function at line 10.

Algorithm 2. Decode a context Id

Input: the encoding id; the function m at which the encoding was emitted; the edge set E; the edge annotations En.

Output: the explicit context cc.

```
1:  Decode (id, m, E, En) {
2:      cc ← "m"
3:      n ← m
4:      while n ≠ root do:
5:          for each e = ⟨p, n, ℓ⟩ in E do:
6:              if En(e) ≤ id < En(e) + numCC[p] then:
7:                  cc ← "p^ℓ" • cc
8:                  id ← id − En(e)
9:                  break
10:         n ← p;
11: }
```

Consider the example in Fig. 4. Assume the ID 6 is generated at function I. The algorithm starts from I. Since $En(GI) \equiv 4 < 6 < 7 \equiv En(GI) + numCC(G)$, the context must have followed the edge GI. The edge is taken and the encoding is decremented by 4 to the value 2. At G, since $En(JG) \equiv 2 \leq 2 < 3 \equiv En(JG) + numCC(J)$, edge JG is taken. Finally, edge AJ is taken, yielding the context AJGI.

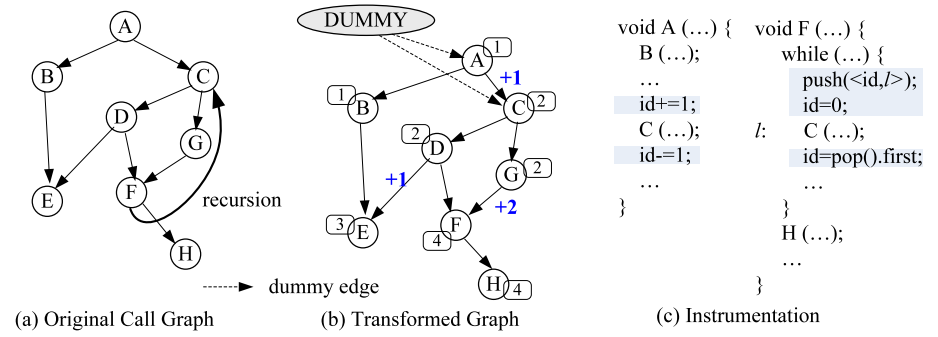


Fig. 5. Example for encoding cyclic CGs.

5 Encoding with Recursion

In the presence of recursion, a context may be of unbounded length, making encoding using a bounded number infeasible. We propose to use a stack to encode contexts with recursion. The basic idea is to encode the acyclic subpaths of a recursive context. When recursion occurs, the current acyclic encoding is pushed to the stack and the following acyclic subpath is encoded with a new ID. The numbers on stack and the current ID together represent the context. In order to perform correct acyclic subpath encoding, recursive CGs need to be transformed.

Our solution is presented in Algorithm 3. The first step deals with CG transformation (in *AnnotateRecursive()*). A dummy root node is introduced. A dummy edge is introduced between the new root and the original root. Dummy edges are further introduced between the new root and any nodes that are the target of a *back edge*, an edge that induces a cycle in the call graph as detected by a depth first search. Note that these edges may differ, depending on how the depth first search is performed, but the algorithm is still correct. Note that only one dummy edge is introduced even if there are multiple back edges to the same node. A dummy edge is always the first edge in the set of incoming edges for a node. In the transformed graph, back edges are removed to allow acyclic encoding. Consider the example in Fig. 5. The back edge FC is removed and dummy edges are introduced between the dummy root and the original root A as well as between the dummy root and the recursive edge target C. Intuitively, after transformation, acyclic subsequences of a recursive context become valid contexts in the transformed CG. Hence, they can be taken into account in the annotation computation. In Fig. 5b, the dummy edge from DUMMY to C makes the acyclic subpaths, such as CDF and CGF, become valid contexts and have unique encodings in the transformed graph. Note that paths that do not involve recursion, such as ACGFH, are not divided into subpaths, even if they contain a node that may be the target of a back edge, such as C in this case.

The instrumentation algorithm is shown in function *InstrumentRecursive()*. The instrumentation is performed on the original graph, which may have back edges. Since the transformed graph shares the same node set as the original graph (except the dummy root), the acyclic node annotations on the transformed graph are also annotations on the original graph and hence are used in instrumentation. Similarly to the previous algorithm, s maintains the sum of contexts of callers that precede the one being processed. At line 12, it is initialized to 1 if the node could be a back edge target, 0 otherwise. Setting to 1 respects the numbering caused by the dummy edge on the transformed graph. Lines 14-17 handle nonback-edges, and they are the same for acyclic graphs. Lines 19-21 handle back edges. Specifically, before a recursive invocation, the current id and the call site are pushed to the stack, and id is reset to 0. Resetting id indicates that the algorithm starts to encode the next acyclic subpath. After the invocation, id is restored to its previous value. Fig. 5c shows the instrumentation for functions A and F, which are the callers of C.

Algorithm 3. Handling Recursion

Description:
$\langle N', E' \rangle$ represents the transformed CG;
$stack$ is the encoding stack.
An edge $e \in E$ is a triple $\langle p, n, \ell \rangle$, where:
$p, n \in N$ are a caller and callee, respectively
ℓ represents a call site where p potentially calls n

```
1: AnnotateRecursive (N, E) {
2:     N' ←{DUMMY} ∪ N
3:     E' ← E ∪ {⟨DUMMY, root, −⟩}
4:     for each back edge e = ⟨p, n, ℓ⟩ in E do:
5:         E' ← E' - e
6:         E' ← E' ∪ {⟨DUMMY, n, −⟩}
7:     Annotate(N', E')
8: }
9: InstrumentRecursive (N, E) {
10:    AnnotateRecursive (N, E)
11:    for n ∈ N do:
12:        s ← (n has a dummy edge in E') ? 1 : 0
13:        for each edge e = ⟨p, n, ℓ⟩ in E do:
14:            if e is not a back edge then:
15:                insert [id = id + s] before ℓ
16:                insert [id = id − s] after ℓ
17:                s ← s+numCC(p)
18:            else:
19:                insert [push(⟨id, ℓ⟩)] before ℓ
20:                insert [id = 0] before ℓ
21:                insert [id = pop().first] after ℓ
22: }
23: DecodeStack (id, stack, m, E', En^E') {
24:    ℓ ←∅
25:    while true do:
26:        Decode (id, m, E', En^E')
27:        wcc ← cc • wcc
28:        if stack.empty() then:
29:            break
30:        ⟨id, ℓ⟩ ← stack.pop()
31:        m ← the residence function of ℓ
32: }
```

Consider the example context ACGFCDE. It is encoded as a subpath with ID 3 on the stack and the current subpath $id = 1$. The encoding 3 is pushed to the stack before F calls C. After the id value is pushed, it is reset to 0. As a result, taking the remaining path CDE leads to $id = 1$. Assume the execution returns from E, D, C, and then calls C, D, F, and H,

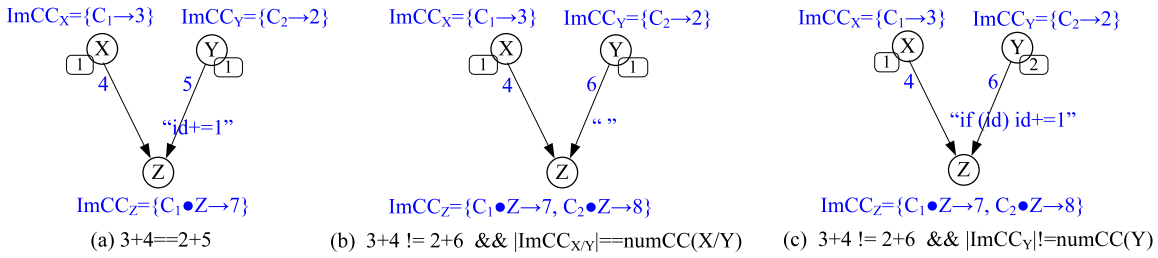


Fig. 6. Intuition of hybrid encoding.

yielding the context of ACGFCDFH. The context is encoded as 3 on the stack and current $id = 0$.

The decoding algorithm for recursive CGs is presented in function *DecodeStack()*. It takes the numbers on the stack, the current subpath encoding, and the current function as part of its inputs. Intuitively, it decodes one acyclic subpath of the recursive context at a time until all encodings on the stack are decoded. Decoding an acyclic subpath is done by calling the acyclic decoding algorithm on the transformed graph at line 26. The resulting acyclic subpath cc is concatenated with the whole recovered context wcc. At line 31, the call site label ℓ is used to identify the method in which the id was pushed to the stack, which is also the starting point of the next round of decoding.

Consider an example. Assume we want to decode a context represented by the stack having an entry $\langle 3, \texttt{F to C}\rangle$, the current $id = 1$, and the current function E. After the first iteration, i.e., *Decode*$(1, \texttt{E}, \ldots)$, the subpath CDE is decoded. Function F is decided as the starting point of the next round of decoding, according to the call site on the stack. After the round, the value 3 on the stack is decoded to the subpath ACGF. The two subpaths constitute the context.

Our design can easily handle very deep contexts caused by highly repetitive recursive calls. In such cases, contexts are a string with repetitive substring patterns such as ACGF CGF CGF CGF . . . in Fig. 5. Such redundancy is not directly removable without relatively expensive compression. With encoding, the repetitive patterns are encoded into repetitive integers and can be further encoded as a pair of ID and frequency. The above context can be encoded to two pairs $3:1$ and $2:3$, with the former representing ACGF and the latter representing the three repetitions of CGF.

6 Safe Hybrid Encoding Leveraging Stack Offsets

The encoding algorithms we have discussed so far explicitly produce a unique ID for each calling context. We call them *explicit encoding* techniques. At runtime, it is often the case that the stack offset, namely, the value of the current stack pointer subtracted by the base of the entire stack, can disambiguate the current context [11]. Consider the example in Fig. 2c. Previously, updates to id had to be inserted on edge CD to distinguish the two contexts of D. Let the stack offset at the entry of D in the context of ABD be x and the offset in the context of ACD be y. If x does not equal y, the two contexts can be disambiguated without any explicit encoding. We call such stack offset-based encoding *implicit encoding* as explicit instrumentation is not needed.

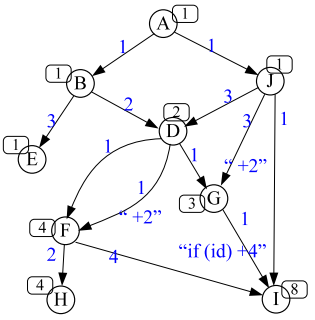
In reality, a few factors make applying implicit encoding difficult. First of all, there may be multiple contexts that map to the same implicit encoding, i.e., they have the same stack offset. Consider the example in Fig. 4. The two contexts $\texttt{ABD}^1\texttt{F}$ and $\texttt{ABD}^2\texttt{F}$ may have the same stack offset because the same sequence of functions are called. Second, programming languages such as C/C++ allow declaring variable-size local arrays. GCC allocates such arrays on stack. Stack allocations make stack offsets variable, so implicit encoding is infeasible. This is further discussed in Section 7. Third, in the presence of recursion, stack offsets cannot be statically reasoned about, which makes it inapplicable.

In order to address the aforementioned issues, we propose a hybrid encoding algorithm that performs explicit encoding when implicit encoding is not applicable. The algorithm is safe, meaning that it uniquely encodes each possible context. The intuition of the hybrid algorithm is presented in Fig. 6. Besides the number of contexts, each node is also annotated with a set of implicit contexts, denoted as $ImCC$, which represents a set of contexts of the node having distinct and fixed stack offsets. It is a mapping from contexts to their offsets. For instance, the implicit context set of node X in Fig. 6 contains context C_1 and its stack offset 3 (symbol $\mapsto$ represents the maps-to relation). Each edge is annotated with two pieces of information. The first piece is the stack frame offset, which is the difference between the stack pointer and the base of the current stack frame when the invocation denoted by the edge occurs. The stack offset of a context can be computed by aggregating the edge offsets along the context. The second annotation is the instrumentation, which is quoted. Symbol “•” represents concatenation.

The figure presents three cases. In case (a), the two implicit contexts of Z have a conflicting offset, namely, $3 + 4 \equiv 2 + 5$. We cannot implicitly encode both. In such a case, we implicitly encode the context along edge XZ and explicitly encode that from YZ. Hence, $ImCC_Z$ is set to contain the context from edge XZ and id is increased by $numCC(X) = 1$ along edge YZ. The instrumentation has the same effect of separating the encoding space as in previous algorithms. In case (b), the two implicit contexts do not conflict and all contexts of X and Y can be implicitly encoded, implied by the subcondition $|ImCC_{X/Y}| \equiv numCC(X/Y)$. In such a case, no explicit encoding is needed and the $ImCC$ set of Z contains both. In case (c), the two implicit contexts of Z do not conflict but the contexts of Y are heterogeneously encoded, denoted by $|ImCC_Y| \neq numCC(Y)$. It implies that id may be nonzero at Y, depending on the context at runtime. If id is not zero, explicit encoding must have been used, and the new context of Z should be explicitly encoded. Hence, id is increased by $numCC(X) = 1$. If id is zero, the context of Y is one of the contexts in $ImCC_Y$. Because the corresponding context of Z does not have conflicts and can be implicitly encoded, no update to id is needed. The above logic is realized by the guarded instrumentation on edge YZ in Fig. 6c.

node	ImCC	
	context	offset
D	ABD	3
	AJD	4
F	ABD^{1}F	4
	AJD^{1}F	5
G	ABDG	4
	AJDG	5
I	ABD^{1}FI	8
	AJD^{1}FI	9
	ABDGI	5
	AJDGI	6
	AJI	2

(a) Implicit contexts



(b) New encoding scheme

context	offset	id
ABD^{1}FI	8	0
AJD^{1}FI	9	0
ABD^{2}FI	8	2
AJD^{2}FI	9	2
ABDGI	5	0
AJDGI	6	0
AJGI	5	6
AJI	2	0
ABDG	4	0
AJDG	5	0
AJG	4	2

(c) Context encodings

Fig. 7. Example of hybrid encoding. Edge instrumentation is quoted and stack frame offsets are not.

The algorithm is presented in Algorithm 4. Variable $ImCC$ represents the set of contexts that are implicitly encoded for each node. Function $extend(ImCC[p], n)$ extends the implicit contexts in $ImCC[p]$ along the invocation $p \rightarrow n$. This is done by concatenating n to the contexts in $ImCC[p]$ and increasing the corresponding stack offsets by the stack frame offset of $p \rightarrow n$. Function $conflict(ImCC_x, ImCC_y)$ tests if two implicit context sets conflict. This is done by checking whether there are two contexts in the two respective sets that map to the same stack offset.

Algorithm 4. Hybrid Encoding

DEFINITIONS:
$ImCC[\,]$ an array of implicit contexts indexed by nodes;
extend $(ImCC[p], n)$=

$$\{C \bullet n \;\mapsto\; t + offset(p, n) \mid C \mapsto t \in ImCC[p]\}$$

in which $offset(p, n)$ is the stack frame offset of the call to n in p.
conflict $(ImCC_x, ImCC_y)$=

$$\begin{cases} 1 & \exists\, C_1 \mapsto t \in ImCC_x \wedge C_2 \mapsto t \in ImCC_y \\ 0 & otherwise \end{cases}$$

An edge $e \in E$ is a triple $\langle p, n, \ell \rangle$, where:
$p, n \in N$ are a caller and callee, respectively
ℓ represents a call site where p potentially calls n

```
1:  Instrument (N, E) {
2:      for n ∈ N in topological order do:
3:          ImCC[n] ← Ø
4:          s ← 0
5:          for each edge e = ⟨p, n, ℓ⟩ in E do:
6:              ImCC′ ←extend (ImCC[p],n)
7:              if not conflict (ImCC[n], ImCC′) then:
8:                  ImCC[n] ← ImCC[n] ∪ ImCC′
9:                  if |ImCC[p]| ≠ numCC[p]) then:
10:                     /*case (c) in Fig. 6, otherwise case (b)*/
11:                     replace the call [ ℓ : n(...) ] in p with
                                [ if (id) {
                                    id = id + s;
12:                                 n(...);
                                    id = id − s;
                                  } else n(...); ]
13:             else:
14:                 /*case (a) */
15:                 insert [ id = id + s ] before ℓ
16:                 insert [ id = id − s ] after ℓ
17:             s ← s+numCC[p]
18: }
```

The instrumentation algorithm is described in function *Instrument()*. Here, we only present the algorithm for acyclic graphs. The extension to recursion can be achieved in a way similar to explicit encoding and hence omitted. The algorithm traverses the nodes in the topological order. It first resets the $ImCC$ for the current node n at line 3. It then traverses the set of invocations to n, denoted by edges of the form $\langle p, n, \ell \rangle$, with p the caller and ℓ the call site. It extends $ImCC[p]$ to n at line 6, and then tests if the extended set conflicts with the implicit contexts of n that have been computed so far. If there is no conflict, the extended set is admitted and aggregated to the implicit set of n at line 8. Line 9 checks if all contexts of p are implicit. If so, the aforementioned case (b) in Fig. 6 is encountered. There is no need to instrument the invocation. If not, case (c) is encountered, namely, the contexts of p may be explicitly encoded. Hence, the instrumentation should decide at runtime if the context has been explicitly encoded. If so, the instrumentation will continue to perform explicit encoding as shown by the boxes in lines 11-12. If the extended implicit set incurs conflict, case (a) is encountered. In lines 15-16, the edge is explicitly encoded.

6.1 Encoding Example

Consider the earlier example in Fig. 4. Fig. 7b shows the graph with stack frame offset annotations and instrumentation (quoted). For instance, when `B` is called inside `A`, the stack frame offset is 1; when `D` is called inside `B`, the frame offset is 2. Hence, the stack offset of the context `ABD` is the sum of the two, which is 3. Similarly, the stack offset of `AJD` is $1 + 3 = 4$. The two different offsets disambiguate the two contexts, and thus the implicit context set of `D`, as shown in Fig. 7a, contains both contexts. No instrumentation is needed.

Now, let us consider `F`. When the first edge is processed, $extend(ImCC[D^1], \texttt{F})$ is computed as $\{\text{ABD}^1F \mapsto 4, \text{AJD}^1F \mapsto 5\}$ at line 6, and it is assigned to $ImCC[F]$ at line 8. When the second edge is processed, $extend(ImCC[D^2], ttF)$ is computed as $\{\text{ABD}^2F \mapsto 4,\ \text{AJD}^2F \mapsto 5\}$. The conflict test at line 7 fails, so $ImCC(F)$ only contains the set extended along the edge D^1F, as shown in Fig. 7a. Moreover, the algorithm instruments the edge D^2F according to lines 15-16.

When `I` is considered, the extensions from $ImCC[F]$, $ImCC[G]$, and $ImCC[J]$ do not conflict. However, contexts to `G` may be explicitly encoded as $|ImCC[G]| = 2 \neq 4 = numCC[G]$. The instrumentation has to be guarded as shown on the edge `GI`. Sample encodings can be found in Fig. 7c. Compared to the instrumentation in Fig. 4, in which five edges need to be instrumented with each instrumentation comprising one addition and one subtraction (two reads and two writes), the hybrid version instruments three edges. Furthermore, the instrumentation on edge `GI` may need just one read (the read in the predicate).

6.2 Decoding Example

The decoding algorithm is elided for brevity. We will use examples to intuitively explain the idea. The encoding of a context under the hybrid algorithm is a triple, which comprises the stack offset, the explicit ID, and the current function. Note that only the explicit ID is computed by instrumentation; the other two can be inferred at any execution point. Assume we are given the encoding of $offset = 5$, $id = 0$ and the current function G. The explicit encoding $id = 0$ means that the context is not explicitly encoded and can be looked up from the $ImCC$ set. From $ImCC$[G], we recover the context as AJDG.

Assume we are given the encoding $offset = 5$, $id = 6$ and the current function I. The nonzero explicit encoding means that explicit encoding is used. From the encoding graph in Fig. 7b, we know that explicit IDs at I in range $[4, 7)$ represent contexts along the edge GI. We reverse both the stack offset and the explicit encoding along this edge and get $offset = 5 - 1 = 4$ and $id = 6 - 4 = 2$. The IDs at G in range $[2, 3)$ represent contexts along JG. Backtracking along the edge leads to $offset = 4 - 3 = 1$ and $id = 2 - 2 = 0$. Now with $id = 0$, we know that the remaining part of the context can be looked up, yielding AJ. Here, we recover the whole context AJGI.

7 Handling Practical Issues

7.1 Handling Insufficient Encoding Space

Since our technique uses a 32-bit ID, it allows a function to have a maximum of 2^{32} different contexts. We observe for some large programs a function may have more than 2^{32} contexts. For example, in GCC, there are a few functions that are called by a few thousand other functions, leading to an overflow in the encoding space. In order to handle such cases, we use a selective reduction approach. We use profiles to identify hot and cold call edges. Cold edges are replaced with dummy edges such that subpaths starting with these cold edges can be separately encoded. As a result, the overall encoding pressure is reduced. At runtime, extra pushes and pops are needed when the selected cold edges are taken.

7.2 Handling Function Pointers

If function pointers are used, points-to analysis is needed to identify the targets of invocations. Due to the conservative nature of points-to analysis, the possible targets may be many. We employ a simple and safe solution. We profile the set of targets for a function pointer invocation with a number of runs. Edges are introduced to represent these profiled targets and then instrumented normally. During real executions, if a callee is not in the profiled set, push and pop are used. In general, we push the id to the encoding stack when an invocation is carried out through a function pointer and pop when it returns. Similarly to how we handle recursion, we introduce dummy edges from DUMMY to functions whose addresses have been explicitly taken and thus may be the potential targets of function pointers. The approach here is based on an overly conservative approximation of the call graph, where any indirect call may lead to any function that can be a target of such a call. Alternatively, a precise points-to analysis could be used to improve efficiency on executions that deviate substantially from the profiled behavior.

7.3 Handling Setjmp/Longjmp

Setjmp allows a developer to mark a position in the calling context that a successive use of longjmp can then automatically return to, unwinding the calling context to the marked point. Our approach safely handles such unwinding by detecting when a setjmp is encountered and storing a copy of the context stack height and current context identifier within the local variables of the function containing setjmp. When longjmp unwinds the calling context, these values are then safely restored from the local copies, unwinding the context encoding as well.

7.4 Handling Stack Allocations

Implicit encoding is not possible when stack allocation is used, e.g., when allocating a variable-size array on the stack. We use static analysis to identify all functions that have stack local allocation and prohibit implicit encoding for those functions.

7.5 Handling Object-Oriented Languages

The approach is presented primarily in the context of an imperative language like C, but it also applies to object-oriented languages like C++ and Java. Exceptions, for instance, are handled in precisely the same way as setjmp/longjmp, which can be used to implement exceptions. Similarly, polymorphic calls behave as indirect calls using function pointers.

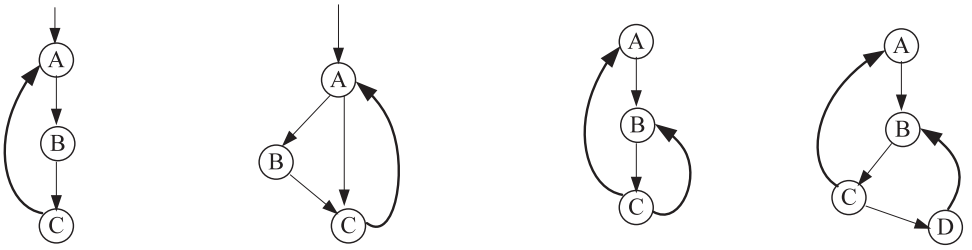
However, there are new behavioral issues that arise. Most notably, object-oriented languages place greater emphasis on indirect calls, namely, through polymorphism. This means that efficiency is more contingent upon how indirect calls are handled. Recall that we handle indirect calls by profiling the runtime behavior, using slower pushing and popping for behaviors that deviate from the profile. Thus, collecting a representative profile is even more important in an object-oriented setting, where such calls are more frequent.

Dynamic loading also presents an issue because the structure of the call graph may not be known a priori. However, by conservatively pushing and popping at the interface boundaries between code loaded at different times, the approach maintains validity. If the code that *may* be loaded can be identified before execution, further optimization can be performed.

Other issues may depend on the environment in which a program executes. For instance, while some Java virtual machines (JVMs) allow stack offsets to be computed [17], this ability is not guaranteed by all JVMs. When executing with a managed runtime environment that prohibits such knowledge, the additional optimizations using hybrid encoding are not feasible.

8 Unrolling Recursion for Better Encoding

Our evaluation in Section 9 shows that the technique proposed in the previous sections is efficient and effective. The overhead is 0-9.8 percent without hybrid encoding and 0-6.4 percent with hybrid encoding. In most cases, it can encode contexts to a very small number of IDs on the stack. However, for some benchmarks such as `186.crafty` and `197.parser`, we need a large number of IDs on the stack (34 and 36, respectively) to encode a context in the worst case due to recursion. Hybrid encoding can never be applied to



(a) Simple recursion (b) Diff. recursion paths (c) Nested recursions (d) Irreducible

Fig. 8. Various forms of recursion.

recursive calls, so the culprit for this inefficiency is the recursion encoding scheme from Section 5. We discuss how to mitigate the problem with that approach in this section.

As discussed in the previous sections, the encoding ID is pushed onto the stack and a new ID is used in two cases: 1) when a backedge is taken, 2) when the target of a function call is not known at compile time due to the use of a function pointer. In practice, the first case is dominant. Most of the programs studied in Section 9 have recursion. However, due to the simplicity of their recursion patterns, we are able to encode their contexts with very few IDs. Fig. 8a shows an example of such simple recursion. The recursion produces a unique path pattern $(\mathrm{ABC})^*$. Since our algorithm encodes the acyclic path `ABC` and uses run length encoding to compress the repetition of the acyclic path, the depth of the stack remains small. However, some programs have complex recursion patterns. Figs. 8b, 8c, and 8d give some examples. Fig. 8b presents recursion with two possible acyclic paths, leading to the runtime path pattern $(\mathrm{AC} \mid \mathrm{ABC})^*$. Note that run length encoding cannot compress paths such as `AC ABC AC ABC AC ABC` as the consecutive acyclic paths have different encodings. Fig. 8c shows nested recursion, which can generate runtime paths such as `ABC BC ABC BC` that are hard for run length encoding. Fig. 8d shows an example with more complex recursion. Indeed, the graph is irreducible.

While it is hard to encode arbitrary recursive paths in general, we observe that in practice even programs with complex recursion often generate regular runtime paths. Unfortunately, the simple run length encoding fails to exploit such regularity. For example, the aforementioned sample paths for Figs. 8b and 8c cannot be compressed, despite their regularity. Observe that if we can unroll the recursion, we can encode the subpath `AC ABC` (for Fig. 8b) as an acyclic path and capture the regularity. However, it is undesirable to actually unroll the recursion because it may substantially increase the code size and disturb the cache behavior at runtime. Next, we propose an encoding algorithm that can achieve the unrolling effect during encoding without physically unrolling the recursion. Our algorithm reduces the maximum encoding stack depth, the largest number of IDs we need to encode a context, from 36 to 13 for `197.parser` and from 34 to 3 for `186.crafty`.

Our algorithm acquires an encoding graph by unrolling a given strongly connected component (SCC). The graph is only used for generating instrumentation; the program itself is never unrolled. The strongly connected component may involve multiple backedges, and such edges may even lead to different nodes. In particular, given a strongly connected component and the number of times c that one expects to unroll, our algorithm first transforms the call graph in the same way as described in Section 5 by replacing backedges with dummy edges from the `DUMMY` node. The number of contexts for each node ($numCC()$) is then computed. After that, SCC is unrolled c times.

In each round of unrolling, a copy of SCC is added to the graph, excluding all backedges. For each backedge $p \rightarrow n$ in SCC, the edge is replaced by a dummy edge from the copy of p in the previous round of unrolling to the new copy of n. Intuitively, the backedge is replaced by an edge from the previous copy of SCC to the current copy, achieving the effect of unrolling.

8.1 Unrolling Example

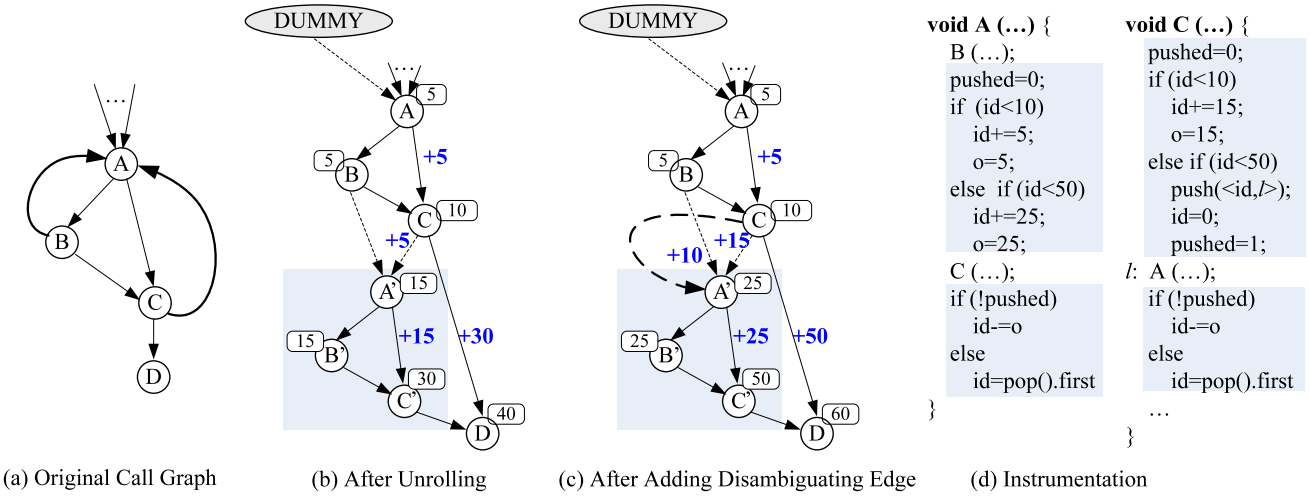
Consider the example in Fig. 9. Nodes `A`, `B`, and `C` and their edges constitute a strongly connected component. Call this SCC. Observe that there are many possible acyclic paths in SCC, and thus run length encoding cannot exploit regular paths such as `AB ABC AB ABC`... Assume we want to unroll SCC once. According to the previous discussion, the graph in (a) is first transformed to the subgraph in (b) that is not highlighted, i.e., the subgraph of nodes `DUMMY`, `A`, `B`, `C`, and `D`. Then, a copy of SCC is added. Here, we use `A'`, `B'`, and `C'` to denote the copies of the corresponding nodes. All edges except backedges are retained, which explains the presence of edge $\mathrm{C}' \rightarrow \mathrm{D}$. Dummy edges (the straight ones) are added from `B` to `A'` and from `C` to `A'`. There is an extra dummy edge from `C` to `A'` (the curved one), which we will explain next.

The aforementioned unrolling is typical. Even though we can now compute the instrumentation based on the unrolled graph, since we are not physically unrolling the code itself, *the instrumentation on the multiple unrolled copies of an edge needs to be properly combined on the single real edge*. In other words, *we need to be able to decide what instrumentation should be applied at runtime.*

8.2 Unrolling Example (Continued)

Fig. 9b shows the unrolled graph and the instrumentation computed based on the algorithm in Section 5. Here, we assume $numCC(A)$ is 5. Observe that edges $\mathrm{A} \rightarrow \mathrm{C}$ and $\mathrm{A}' \rightarrow \mathrm{C}'$ have different instrumentation, specifying the different increments to the ID according to the different contexts of the edge. In particular, the ID shall be incremented by 5 and by 15 when the edge is encountered for the first and the second time of the recursion, respectively.

One possible solution is using a counter to keep track of the number of iterations and then using the counter to select instrumentation executed at runtime. However, maintaining such a counter is nontrivial. It needs to be initialized the first time a strongly connected component is entered. The initialization may be complex if there are multiple entry points, in which case the counter may need to be initialized



(a) Original Call Graph (b) After Unrolling (c) After Adding Disambiguating Edge (d) Instrumentation

Fig. 9. Example for unrolling cyclic CGs.

at all such places. Furthermore, the counter needs to be incremented when a backedge is taken, and it needs to be reset to 0 when the unrolling bound is met. For each edge in the recursion, we also need to allocate an array to remember the different runtime actions.

We propose a more sophisticated and efficient solution. We use the current ID value to disambiguate the different contexts of an edge such that the corresponding instrumentation can be applied. The idea is intuitive because the ID value indeed encodes the current context. However, the unrolling transformation discussed so far is not sufficient. Consider the unrolled graph in Fig. 9b. Assume in one case the path `ACD` is taken and in the second case the path `ABCA'B'C'D` is taken. Both edges $C \rightarrow D$ and $C' \rightarrow D$ correspond to the call to function `D()` inside function `C()`. Note that these two edges have different instrumentation, and we need to decide which should be applied when `D()` is called. However, the current ID value is 5 at both `C` in case one and `C'` in case two. From the value it is impossible to disambiguate the needed instrumentation.

To solve the problem, we introduce dummy edges, called the *disambiguating edges*, from the node in the previous unrolled copy (or the original recursion) that has the maximum number of contexts to each of the nodes in the current unrolled copy that are the target of a backedge. We also force a disambiguating edge to be the first edge in the incoming edge set. Suppose one such edge is $p \rightarrow n$. Intuitively, the disambiguating edge does not correspond to any real call edge, which is different from other dummy edges, introducing it as the first edge of n forces the ID to be incremented by an additional $numCC(p)$ for all the other edges of n.[1] As a result, all possible ID values for any nodes in the previous unrolled copy must be smaller than $numCC(p)$, and all possible ID values in the current unrolled copy must be larger or equal to $numCC(p)$. Hence, we can use $numCC(p)$ to perform safe disambiguation.

1. Recall that the basic algorithm in Section 4 dictates that the ID incrementation on the ith edge of n is the aggregation of the $numCC()$s of n's predecessors along the first to the $(i-1)$th edges.

8.3 Unrolling Example (Continued)

Since, in the original copy of SCC, node `C` has the most contexts, in Fig. 8c, a disambiguating edge is added from `C` to `A'` (the curved edge). As a result, the ID instrumentation on $B \rightarrow A'$ and $C \rightarrow A'$ is increased by $numCC(C) = 10$ compared to that in (b). As a result, all nodes in the original SCC must have ID values smaller than 10 and all nodes in the unrolled copy must have ID values larger or equal to 10. The code in (d) partially shows the instrumentation. Consider function `A()` first. According to the encoding graph in (c), the two edges $A \rightarrow B$ and $A' \rightarrow B'$ have no instrumentation. Hence, the call site to `B()` is not instrumented. In contrast, the two edges corresponding to the call to `C()` have different instrumentation that is determined by the predicates at the two conditional statements before the call site. Note that the values 10 and 50 are indeed $numCC(C)$ and $numCC(C')$. They are the maximum numbers of contexts in the original SCC and the unrolled copy, respectively. Variable $pushed$ is used to record if the executed instrumentation was a push to the encoding stack; o is used to record the increment to the ID. The instrumentation after the call `C` restores the original ID value. The instrumentation for function `C` is similar. The only difference is that the second `if` statement guards a push to the stack, corresponding to `C'` calling `A` at the end of the unrolled copy.

As a result, the path `AB ABC` has the encoding 10, and hence regular paths `AB ABC AB ABC AB ABC` can be compressed to $10 : 3$ by run length encoding.

The encoding technique is presented in Algorithms 5 and 6. Method *Unrolling()* describes the unrolling transformation. It first removes all backedges and introduces dummy edges by calling *AnnotateRecursive()* (line 2). The main unrolling process is described in the loop in lines 3-10. For each unrolling iteration, a copy of SCC is added excluding the backedges (lines 4-5). Dummy edges are introduced between callers in the previous copy and the corresponding callees in the current copy to replace backedges (lines 6-7). Then, disambiguating edges are added (lines 8-9). At the end of each iteration (line 10), the new graph is annotated again to recompute the number of contexts because such information will be used in the next round of unrolling. Finally, after the c rounds of unrolling, edge instrumentation is computed by calling method *MarkEdge()* (line 11). Method *MarkEdge()*

computes edge marks (e.g., those in Fig. 9c) in the same way as edge instrumentation was computed in Section 4.

Algorithm 5. Unrolling Recursion

Description:
$\langle N', E' \rangle$ represents the transformed CG;
An edge $e \in E$ is a triple $\langle p, n, \ell \rangle$, where:
$p, n \in N$ are a caller and callee, respectively
ℓ represents a call site where p potentially calls n
SCC represents the strongly connected component;
c the number of unrollings;

```
 1: Unrolling (N, E, SCC, c) {
 2:     AnnotateRecursive (N, E)
 3:     for i = 1 to c do:
 4:         N' ← N' + nodes in SCC
 5:         E' ← E' + edges in SCC except backedges
 6:         for each backedge e = ⟨p, n, ℓ⟩ in SCC do:
 7:             Add a dummy edge from p in the (i − 1)th
                unrolled copy to n in the ith copy.
 8:         for each node n ∈ SCC that is the target of
            backedge do:
 9:             Add a dummy edge from the node with the
            maximum numCC() in the (i − 1)th copy to n in the ith
            copy and make it the first incoming edge of n.
10:         Annotate (N', E')
11:     MarkEdge (N', E')
12: }
13:
14: MarkEdge (N, E) {
15:     for n ∈ N do:
16:         s ← 0
17:         for each e = ⟨p, n, ℓ⟩ in E do:
18:             Mark[e] = s
19:             s ← s + numCC[p]
20: }
21:
```

InstrumentUnrolled() (in Algorithm 6) is the instrumentation procedure. It first performs unrolling (line 23). Note that although the algorithm is for unrolling one strongly connected component, it can be easily extended to unroll multiple such components. The main instrumentation loop (lines 24-44) iterates over the nodes in the original graph instead of the unrolled encoding graph. For each node, the algorithm iterates over its real edges (lines 25-44). If the edge is not part of the strongly connected component, it instruments the edge (lines 38-44) in a way very similar to method *InstrumentRecursive()* in Algorithm 3. Otherwise, the algorithm first inserts the initialization of the variable $pushed$ (line 27), which indicates whether an entry is pushed to the stack. The loop on lines 28 to 34 integrates the instrumentation for the various copies of the edge. If the edge is a backedge and we are processing the last unrolled copy (line 30), meaning that the path is about to exit the unrolled region, instrumentation is inserted before the call site to push the current ID onto the stack, reset the ID value, and set the variable $pushed$ (line 31). Otherwise, the copy of the edge in the current copy of SCC is identified (line 33). Instrumentation is added to increment the ID according to the edge mark and the edge mark is stored to variable o to allow later ID recovery (line 34). Line 35 inserts an empty body to the last *else* instrumentation branch to make it syntactically correct. Finally, instrumentation is added after the call site to restore the previous ID value (line 36).

Algorithm 6. Unrolling Recursion (Continued)

```
An edge e ∈ E is a triple ⟨p, n, ℓ⟩, where:
    p, n ∈ N are a caller and callee, respectively
    ℓ represents a call site where p potentially calls n
22: InstrumentUnrolled (N, E, SCC, c) {
23:     Unrolling (N, E, SCC, c)
24:     for n ∈ N do:
25:         for each edge e = ⟨p, n, ℓ⟩ in E do:
26:             if e ∈ SCC then:
27:                 insert [ pushed = 0; ] before ℓ
28:                 for i=0 to c do:
29:                     x ← maximum numCC() in the ith copy of SCC.
30:                     if e is a backedge and i ≡ c then
                            insert [ if(id < x){
                                       push(⟨id, ℓ⟩);
                                       id = 0;
                                       pushed = 1;
                                     } else ] before ℓ
32:                     else
33:                         e' ← the ith unrolled copy of e in E'.
                            insert [ if(id < x)
                                       id += Mark[e'];
                                       o = Mark[e'];
                                     else ] before ℓ
35:                 insert [ {} ] before ℓ
                    insert [ if(!pushed)
                                 id -= o;
                             else
                                 id = pop().first ] after ℓ
37:             else /* e ∉ SCC */
38:                 e' ← the correspondence of e in E'.
39:                 if e' is a dummy edge from DUMMY then
                        insert [ push(⟨id, ℓ⟩);
                                 id = 0; ] before ℓ
                        insert [ id = pop().first ] after ℓ
42:                 else
                        insert [ id += Mark[e']; ] before ℓ
                        insert [ id -= Mark[e']; ] after ℓ
45: }
```

The decoding algorithm is the same as before, using the unrolled encoding graph (e.g., Fig. 9c). Details are omitted.

9 Evaluation

We have implemented our approach in OCaml using the CIL source-to-source instrumentation infrastructure [18]. The implementation has been made available on our project website [12]. All experiments were performed on an Intel Core 2 2.1 GHz machine with 2 GB RAM and Ubuntu 9.04.

Note that because we use source level analyses for our implementation, we do not have as much information as would be available were the analysis within the actual compilation phase. In particular, we do not know if the compiler eventually inlines a function, resulting in indistinguishable stack frame sizes. Hence, we disabled inlining during our experiments. Observe that this is not a limitation of our algorithm but rather an outcome of our infrastructure selection. An implementation inside a compiler after functions are inlined can easily handle the issue.

Table 1 presents the static characteristics of the programs we use. Since CIL only supports C programs, our implementation currently supports C programs. We use SPECint 2000 benchmarks and a set of open source programs. Some of them are large, such as `176.gcc`, `alpine`, and `vim`.

TABLE 1
Static Program Characteristics

programs	LOC	CG nodes	CG edges	recursions	fun pointers	our max ID	BL max ID
cmp 2.8.7	6681	68	162	0	5	44	156
diff 2.8.7	15835	147	465	6	8	645	3140
sdiff 2.8.7	7428	90	281	0	5	242	684
find 4.4.0	39531	567	1362	28	33	1682	5020
locate 4.4.0	28393	320	688	3	19	251	1029
grep 2.5.4	26821	193	665	17	10	17437	44558
tar 1.16	58301	791	2697	19	46	1865752	4033519
make 3.80	29882	271	1294	61	7	551654	1543113
alpine 2.0	556283	2880	26315	302	1570	4294967295*	4.5e+18
vim 6.0	154450	2365	15822	1124	27	4291329441*	8.7e+18
164.gzip	11121	154	426	0	2	536	1283
175.vpr	29807	327	1328	0	2	1848	13047
176.gcc	340501	2255	22982	1801	128	4294938599*	9.1e+18
181.mcf	4820	93	208	2	0	6	96
186.crafty	42203	179	1533	17	0	188589	650779
197.parser	27371	381	1676	125	0	2734	14066
255.vortex	102987	980	7697	41	15	4294966803*	1.5e+13
256.bzip2	8014	133	396	0	0	131	609
300.twolf	49372	240	1386	9	0	1051	3766

**Selective reduction is applied to 288 nodes in `alpine 2.0`, 300 in `176.gcc`, 33 in `255.vortex`, and 877 in `vim`.*

Three SPECint programs, `252.eon`, `254.gap`, and `253.perlbmk`, are not included because CIL failed to compile them due to their use of C++ or unsupported types. For each program, the table also details lines of code (LOC), the number of nodes in the call graph nodes, the number of edges in the call graph edges, the number of recursive calls or back edges in the call graph (recursions), the number of function pointer invocations, and the maximum ID required in the call graph under the original BL numbering scheme (BL max ID) and our scheme (our max ID).

We can observe that most programs make use of recursion and function pointers. In particular, `176.gcc` has 1,800 recursive invocations and uses function pointers at 128 places. We are able to encode the contexts of all programs smaller than 100K LOC in a 32-bit ID. For the larger programs, overflows are observed and selective reduction (Section 7) is performed to reduce encoding pressure and fit the ID into 32 bits. The numbers below the table show the number of nodes on which selective reduction is performed for each large programs. Observe, the maximum ID for the original BL encoding scheme is often a few times larger than ours.

Fig. 10 illustrates the runtime overhead of our encoding algorithms. We use reference inputs for SPEC programs and random inputs for the rest. In particular, we use training inputs to profile SPEC programs. The runtime for `alpine 2.0` cannot be accurately measured as it is an interactive program. The slow down is not humanly observable though. The times for `vim` were collected in batch mode. We normalize the runtime of two encoding algorithms: One is the basic algorithm that handles recursion and the other the hybrid algorithm. From the figure, we observe that our technique is very efficient; the overhead is 0-9.8 percent or 0-6.4 percent for the two respective algorithms. The hybrid algorithm improves over the basic algorithm by 48.1 percent. `176.gcc` and `255.vortex` have relatively higher overhead due to the extra pushes and pops caused by selective reduction.

Table 2 presents the dynamic properties. The second and third columns compare our encoding stack depth (ours) to the plain calling stack depth (plain), i.e., the call path length. The third and fourth columns show the stack size needed to cover 90 percent of contexts at runtime. The last column presents the number of unique contexts encountered. We observe that our maximum encoding stacks are substantially

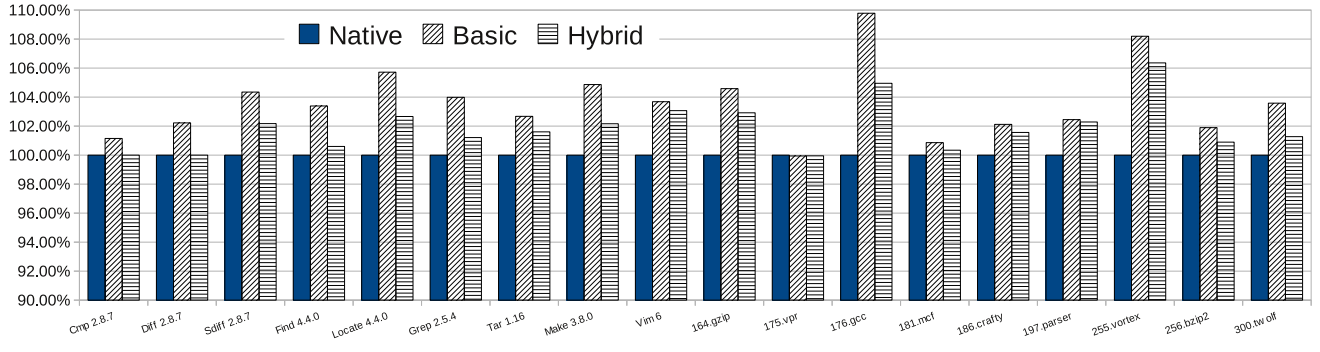


Fig. 10. Normalized runtime comparison of benchmarks with no instrumentation, basic encoding instrumentation, and the hybrid encoding instrumentation. Times are normalized against the native runtime.

TABLE 2
Dynamic Context Characteristics

programs	Max Depth		90% Depth		dynamic
	ours	plain	ours	plain	contexts
cmp 2.8.7	0	3	0	3	9
diff 2.8.7	0	7	0	5	34
sdiff 2.8.7	0	5	0	4	44
find 4.4.0	2	12	1	12	186
locate 4.4.0	0	9	0	9	65
grep 2.5.4	0	11	0	8	117
tar 1.16	3	40	2	31	1346
make-3.80	6	82	3	43	1789
alpine 2.0	11	29	6	18	7575
vim 6.0	10	31	5	10	3226
164.gzip	0	9	0	7	258
175.vpr	0	9	0	6	1553
176.gcc	19	136	2	15	169090
181.mcf	14	42	0	2	12920
186.crafty	34	41	10	23	27103471
197.parser	36	73	11	28	3023011
255.vortex	7	43	2	12	205004
256.bzip2	1	8	0	8	96
300.twolf	4	11	0	5	971
Average	8.78	39.22	2.17	13.72	1795292

shorter than the corresponding maximum plain stacks. For most utility programs, the encoding stack is empty, meaning the contexts can be encoded into one ID without using the stack. We also observe that some programs need maximum encoding stacks with a nontrivial depth, e.g., `176.gcc`, `181.mcf`, `186.crafty`, and `197.parser`. However, when we look at the 90 percent cutoffs, `176.gcc` and `181.mcf` require encoding stacks of depth 2 and 0, respectively. To precisely evaluate our technique, we also contrast the full frequency distributions for our encoding stacks and the corresponding plain stacks. Due to space limits, we only present some of the results in Fig. 11. Each diagram corresponds to one of the benchmarks considered. The x-axis corresponds to stack depth and the y-axis shows the cumulative percentage of dynamic context instances during execution that can be represented in a given stack depth. Thus, if a curve ascends to 100 percent very quickly, it means that most contexts for that benchmark could be stored with a small encoding stack, possibly even size zero. The graph for `164.gzip` is very typical for medium-sized programs. Our stack is always empty while the plain stack gradually ascends. Observe the diagram of `176.gcc`. Even though the maximum encoding stack has the size of 19, our curve quickly ascends over the 90 percent bar with the stack depth 2. The diagrams for other large programs such as `255.vortex`, `vim`, and `alpine 2.0` are similar. Finally, the diagram of `197.parser` shows that our technique is relatively less effective. The reason is that parser implements a recursive descent parsing engine [19] that makes intensive recursive calls based on the syntactic structure of input. The recursion is mostly irregular, so our simple compression does not help much. `186.crafty` is similar.

Unrolling recursion. In this experiment, we evaluate the unrolling technique introduced in Section 8. According to the technique, we unroll the strongly connected components in the encoding graph, and then the instrumentation on the same call edge is integrated through conditional statements. We use the SPEC programs in Table 2 whose maximum encoding stack depths are not small, namely, `181.mcf`, `186.crafty`, `197.parser`, and `300.twolf`. We exclude `176.gcc` and `255.vortex` because they are under high encoding pressure, that is, the static maximum IDs needed to encode all possible contexts exceed 2^{32} without selective reduction. Note that unrolling creates extra encoding pressure. In the experiment, we first identify the executed strongly connected components from the profiles. We exclude all the SCCs that are trivial, meaning self-recursive functions that have only one recursive call site, because they always generate repetitive call paths that can be compressed by run length encoding. Note that a self-recursive function with multiple call sites to itself is not excluded because recursive paths through different combinations of call sites should be considered different contexts. We unroll each SCC a few times according to the profile.

The results are presented in Table 3. The columns labeled with "max depth" compare the maximum encoding stack

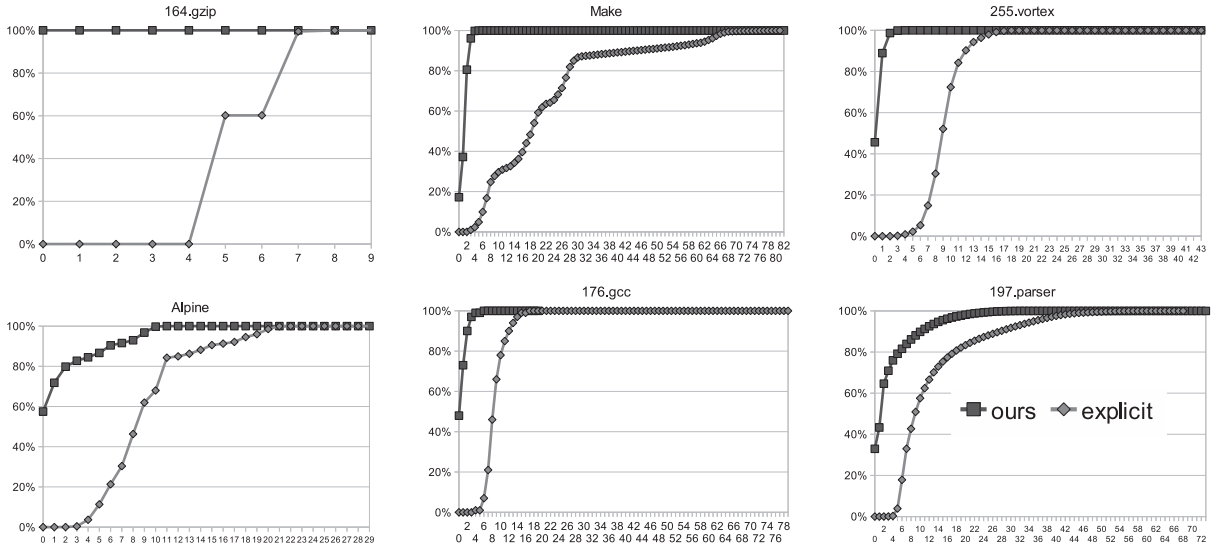


Fig. 11. Context encoding stack size distributions.

TABLE 3
Results After Unrolling Recursion

programs	max depth			max ID		normalized runtime		unrolled SCCs	added nodes	avg. unrollings
	plain	before	after	before	after	before	after			
181.mcf	42	14	3	6	162	1.01	1.029	1	4	4.0
186.crafty	41	34	1	188589	82484285	1.023	1.055	2	4	2.0
197.parser	73	36	4	2734	88334	1.024	1.048	15	24	1.3
300.twolf	11	4	0	1051	21243	1.037	1048	5	12	2.4

depths. In particular, the original calling context depth (column "plain"), the depth using the normal encoding without unrolling ("before"), and the depth after unrolling ("after") are compared. The columns labeled with "max ID" present the maximum encoding IDs needed to encode all contexts "before" and "after" unrolling. The "normalized runtime" columns show the overheads normalized against the native runtime. The "unrolled SCCs" column presents the number of SCCs that are unrolled. The "added nodes" present the number of nodes being added to the encoding graph due to unrolling. The last column shows the average number of unrollings performed. The encoding of the unrolled graph is based on the basic encoding algorithm, without hybrid encoding or selective reduction.

From the results, we observe that unrolling can significantly reduce the maximum runtime encoding stack depths while it induces extra encoding pressure indicated by the increasing the maximum encoding IDs needed. For instance, in benchmark `186.crafty`, after unrolling two SCCs twice, the maximum encoding depth at runtime is reduced from 34 to 1. Also observe that the static maximum encoding ID has also substantially increased due to unrolling. Similar results are observed for `197.parser`. The implication is that if an SCC is nested deep in the original call graph with a large number of possible contexts (e.g., close or exceeding 2^{32}) leading to it, it is unlikely that a 32-bit ID is sufficient for encoding the unrolled graph without selective reduction.

The runtime overhead also increases as a result of unrolling. For three of the benchmarks, the new encoding incurs roughly 5 percent overhead. Note that this is not contradictory to the observation in our previous experiment (Table 2) that 90 percent of contexts encountered at runtime have small encoding depths. In fact, the functions that are unrolled are mostly hot functions even though they rarely produce large encoding depths at runtime.

Finally, all the programs require a small number of unrollings.

10 Client Study: Tracking Variable Definition Context for Debugging

In this section, we propose a particular use for context encoding. A very common practice in debugging crash-based failures is investigating the backtrace of the crash, which is essentially the calling context of the crash point. While the programmer hopes that the root cause resides in one of the functions in the backtrace, in reality this is often not true. The reason is that a crash is often caused by memory corruption, such as nullifying a pointer by mistake. However, a crash does not occur immediately after the corruption. Instead, it often occurs later, when the corrupted memory is referenced. In other words, when the crash occurs, the relevant context may be long gone, and investigating the current backtrace is fruitless.

One possible way to mitigate the problem is by tracking the definition point of each variable dynamically such that when the corrupted variable is referenced, the instruction that corrupts the variable can be identified. This can be done as follows: When an instruction at pc writes to a memory address m at runtime, use a hashtable H to record that the last write to m is from pc, i.e., $H(m) = pc$; later, when m is referenced, pc can be retrieved from H. This is essentially a standard way to detect data dependence. However, it is often insufficient because the corruption point is usually not the root cause.[2] In many cases, corruptions even occur inside library code which is not buggy. Instead, they occur because the library code is called with faulty parameters or the preconditions of the library functions are not checked by the user code.

The observation is that *we should inspect the calling context of the corruption point* instead of that of the crash point. The challenge is that we cannot predict whether a definition is actually corrupting some variables when the definition is performed. For instance, when a buffer is written to through a pointer, it is hardly known if the write is corrupting some other variable. With calling context encoding, *we propose to track both the definition point of a variable and the context of the definition.* In particular, when an instruction at pc writes to a memory address m at runtime and the current context is c,

$$H(m) = (pc, c).$$

When a crash occurs, we can easily inspect the calling context of the corruption point. Note that such tracking is performed for each definition during program execution. In other words, without context encoding, upon each memory write we have to record the current context in its plain form, which incurs prohibitively high time and space overhead. With encoding, only two extra writes, i.e., writing pc and c to the hash table, are usually needed upon each memory write. Each variable needs only two extra 32-bit values unless the context requires a compact stack. In other words, the space overhead is roughly two times what it would be if we tracked definitions at the word level.

Example. `Mplayer-1.0pre6` is a movie player for Linux that plays video in various formats. It has a buffer overflow fault that may cause it to crash when a 2-byte parameter in the audio header of a video file has high values [20]. The code relevant to the bug is shown in

2. Here, we define the root cause of a failure as the faulty semantics in the buggy program that caused the failure.

```
File: libmpcodecs/dec_audio.c
int decode_audio (sh_audio_t *sh_audio,unsigned char *buf,int minlen,int maxlen) {
357   int declen;
        ……
369   declen = af_calc_insize_constrained (sh_audio->afilter,minlen,maxlen,
370     sh_audio->a_buffer_size - sh_audio->audio_out_minsize);          // Root cause.
        ……
382     int len = declen - sh_audio->a_buffer_len;
        ……
390     len = mpadec->decode_audio (sh_audio,
391       sh_audio->a_buffer+sh_audio->a_buffer_len, len, maxlen);
        ……
}

File:  libmpcodecs/ad_pcm.c
static int decode_audio (sh_audio_t *sh_audio,unsigned char *buf,int minlen,int maxlen) {
 95   int len = sh_audio->channels*sh_audio->samplesize-1;
 96   len = (minlen + len) & (~len); // sample align
 97   len = demux_read_data (sh_audio->ds,buf,len);
        ……
}

File: libmpdemux/demuxer.c

// 39 call sites to  demux_read_data()
int demux_read_data (demux_stream_t *ds, unsigned char* mem, int len){
 463    int x = ds->buffer_size - ds->buffer_pos;
        ……
 467    if(x>len) x = len;
 468    if(mem) memcpy (mem+bytes,&ds->buffer[ds->buffer_pos],x);     // Corruption
        …….
}
```

Fig. 12. The relevant code snippet for the `mplayer-1.0pre6` buffer overflow bug. Shaded variables denote the propagation path of the fault.

```
Crash backtrace:
=================
Program received signal SIGSEGV, Segmentation fault.
0x08084adc in config (...) at vo_xv.c:269
269             XGetWindowAttributes(mDisplay, … ,
1: mDisplay = (Display *) 0x8635700

(gdb) bt
#0  0x08084adc in config (....) at vo_xv.c:269
#1  0x080b0af3 in config (....) at vf_vo.c:48
#2  0x080aa7d3 in mpcodecs_config_vo (....) at vd.c:312
#3  0x080acbbc in init_vo (....) at vd_ffmpeg.c:512
#4  0x080acf59 in get_buffer (....) at vd_ffmpeg.c:563
#5  0x082cc770 in cinepak_decode_frame (....) at cinepak.c:396
#6  0x081a8f60 in avcodec_decode_video (....) at utils.c:593
#7  0x080ad256 in decode (....) at vd_ffmpeg.c:765
#8  0x080a9fc1 in decode_video (....) at dec_video.c:309
#9  0x080549af in main (....) at mplayer.c:2340
```

Fig. 13. The backtrace for the mplayer crash.

Fig. 12. The root cause is in the function `af_calc_insize_constrained()` called at line 369 in `decode_audio()` in `libmpcodecs/dec_audio.c`. It miscalculates the buffer size `declen`. The faulty value is propagated to line 391 and then to function `decode_audio()` in `libmpcodecs/ad_pcm.c`. Inside the function, variable `len` has a faulty value, which is further passed to function `demux_read_data()`. At line 468, the buffer pointed to by "`mem+bytes`" is overflowed by the memory copy operation.

However, the program does not crash right away as the result of the overflow. Instead, the crash occurs after some time, in a completely irrelevant context. Fig. 13 shows the backtrace for the crash point. The crash happens because pointer `mDisplay` has a corrupted value in function `config ()` inside file `vo_xv.c`. Observe that the calling context of the crash point provides very little help to locate the root cause. With the proposed technique, we record the definition point of each memory location and its context encoding. In this case, the definition point is found to be inside function `demux_read_data()` because `mDisplay` was defined by mistake due to the overflow. The decoded

```
Corruption point and its context
=================
#0 ... in demux_read_data(....) at demuxer.c:468  //Corruption
#1 ... in decode_audio (....) at ad_pcm.c:97
#2 ... in decode_audio (....) at dec_audio.c:391    //Root cause residence
#3 ... in main (....) at mplayer.c:2267
```

Fig. 14. The backtrace for the definition point of `mDisplay`, which is essentially the corruption point.

TABLE 4
The Results on Real-World Crash-Based Bugs

program	bug ID	corrupting call sites	distance to corruption	crash context relevant?
pine-4.44	[21]	13	2	no
squid-2.3	[22]	1	0	no
bc-1.06	[23]	2	1	no
mplayer1.0pre6	[20]	39	2	no
chmlib0.36	[24]	11	1	no

context is shown in Fig. 14.[3] Observe that from the corruption point and the context, the programmer can easily locate the fault. Knowing the corruption point alone is not sufficient because function `demux_read_data()` can be called at 39 places (and only one context is failure inducing). This case clearly shows the benefits of context encoding.

Study on other crash bugs. We applied our technique to a set of real-world bugs collected from various sources. The results are presented in Table 4. In particular, the first column shows the buggy programs. Bug descriptions can be found through the references in the second column. They are all related to memory corruption and crashing. The third column shows the number of call sites to the function where the corruption occurs. This illustrates that having the corruption points is not sufficient and calling contexts are needed. The fourth column shows the distance between the root cause location and the corruption point. It denotes the number of functions the programmer has to inspect backward along the calling context until he/she locates the root cause. The last column presents whether the calling context of the crash point is relevant.

From the results, we observe that for all these bugs, the contexts of the crash points are not useful. In other words, using the *backtrace* command in gdb does not provide much help. Moreover, for most bugs, the function where the corruption occurs can be called in multiple places, leading to different possible contexts. As a result, having the corruption point alone is not enough. Finally, the contexts of the corruption points often provide good guidance to the faults because the programmer only needs to inspect a few functions.

Note that the technique we propose here is not an autonomous debugging technique but rather a supplemental tool that facilitates the debugging process and should be used in an interactive way. In general, the context where a variable is defined may not provide the precise source of a corrupted value. For instance, the value may be copied in transitively from an originally corrupted variable that is no longer live in the memory. If the corrupted variable is still live, its defining context will lead the programmer one step closer to the root cause.

3. Note that our encoding technique is implemented at source code level and hence only line numbers are available in the decoded context but not the pcs.

11 Related Work

11.1 Explicit Context Encoding

The most direct approach to identifying calling contexts is explicitly recording the entire call stack. There are two main approaches for doing so. *Stack walking* involves explicitly traversing the program stack whenever a context is requested in order to construct the full identifier for the context [25]. Compared to our approach, stack walking is more expensive and hence less desirable. Maintaining the *calling context tree* [3], [5], [26] involves explicitly unfolding a CG into a tree at runtime, with each tree path representing a context. Tree nodes are allocated and maintained on the fly. The current position in the tree is maintained by a traversal action at each call site. While calling context trees are very useful in profiling, our technique is more general as it provides more succinct representation of contexts and has less overhead. Moreover, our encoding technique is complementary to call trees because a compressed tree can be constructed using encoding by denoting tree subpaths as individual nodes.

A more in-depth look at how calling context trees may be affected by particular techniques for acquiring a call graph, selecting which call-edges to consider or combine, and handling recursion may be found in [27].

11.2 Path Encoding

In [13], Ball and Larus developed an algorithm to encode control flow paths. Our technique is inspired by theirs. In comparison, our algorithm is more compact for context encoding as it encodes in a different (backward) direction. The BL algorithm breaks loop paths at control flow back edges and our algorithm similarly separates recursive contexts into acyclic subsequences and encode independently. However, due to the characteristics of contexts, our recursive encoding relies on a stack. Furthermore, we safely leverage stack offsets to remove unnecessary encoding. In [28], the BL algorithm was extended to encode control flow paths across function boundaries. In [29], Law and Rothermel's approach for encoding whole program paths [15] is used to compress function call traces. These are then used to recreate call stacks for further analysis. In [30], it was first proposed that the BL algorithm can be extended to encode calling contexts. However, the approach was not fully developed. It does not encode/decode recursive contexts. No empirical results were reported.

In [31], edge profiles are used to avoid considering cold paths and to infer paths without numbering where possible. If only a subset of paths are known to be of interest, Vaswani et al. [32] use this information to construct minimal identifiers for the interesting paths, while allowing uninteresting paths to share identifiers when advantageous. Similarly to these approaches, our technique uses profiling to guide instrumentation, improve efficiency, and reduce encoding space pressure. If it were known that only a subset of calling contexts were of interest, we could further reduce both our instrumentation and identifier size, but we leave this open as future work.

11.3 Probabilistic Contexts

It may be acceptable that context identifiers are not unique. That is, some arbitrary contexts may be merged with low probability. In such scenarios, *probabilistic calling contexts* can be used to very efficiently identify calling contexts with high probability of unique identifiers. In [6], calling contexts are hashed to numeric identifiers via hashes computed at each function call. Decoding is not supported in this technique. More recently, Mytkowicz et al. [11] use the height of the call stack to identify calling contexts and mutates the size of stack frames to differentiate conflicting stack heights with empirically high probability. Our approach also uses the height of the call stack to disambiguate calling contexts, but in contrast, we only use it to eliminate instrumentation and path numbering where it can be shown that it is safe to do so. Their decoding is achieved by offline dictionary lookup. The dictionary is generated through training, and hence the actual calling contexts cannot always be decoded. In contrast, our approach guarantees safety and easy decoding. Furthermore, their approach is unsafe with stack allocation.

Most recently, the hashing approach from [6] was extended in [10] to enable some limited ability to decode contexts. In contrast to the approach we present, Breadcrumbs, the extended technique, can provide a more efficient encoding in the presence of dynamic class loading, whereas the efficiency of our technique can degrade in such a scenario. However, Breadcrumbs works by capturing extra information as the program executes to guide a search over possible calling contexts. This gives it lower efficiency than our technique in general, and the decoding process can be slow or even fail.

11.4 Context Reconstruction

In [33], a set of partial traces can be used to reconstruct contexts with high probability. Observe that when partial traces are already being collected, this can be of substantial utility, but tracing alone is already more costly than calling context identification. Furthermore, the approach cannot provide a guarantee for the correctness of the reconstructed contexts.

11.5 Sampling-Based Profiling

Counter-based sampling [34] can be used to collect approximate runtime profiles at discrete intervals. Adaptive sampling [7] is used to guide context sensitive profiling, thus reducing the number of times contexts are even needed. While this is useful when performing hot-context profiling, it does not generalize to other uses of contexts where coverage guarantees are important.

11.6 Static Contexts

While this paper considers the precise dynamic calling contexts encountered in an execution, existing work has also examined different *static* notions of context [35], [36], [37], [38] Most similar are k-CFA contexts that encode a bounded length of a calling context or control flow context [37]. When the analysis and instrumentation are performed on a source code level, our work is analogous to a compressed, dynamic, fully precise version of these contexts. Agesen's CPA applies to languages with parametric polymorphism; it differentiates instances, or contours, of polymorphic functions by the types of their arguments [35]. When our analysis is performed over a contour disambiguated call graph and IR instead of on source, it computes the precise, dynamic CPA context instead of just the source-level calling context. In contrast, our approach is unrelated to notions of context that disregard call sites, such as object sensitivity [36].

Analyses using static context approximations can also aid our technique in the future. For example, recent work starts out with crude static context approximations and refines those approximations in response to feedback from a target analysis, allowing for scalable, targeted precision for static analyses [38]. Applying this to the call graphs used in our work, when the encoding space is insufficient we presently revert to pushing and popping (Section 7). Instead, we could use a context sensitive call graph and refine the graph based on the observed encoding pressure. This could allow the finite encoding space to be handled in a more efficient manner, similarly to the approach for unrolling recursion. Investigating this approach is left to future work.

11.7 Others

While we have presented a debugging application of context encoding, other existing fault localization and debugging techniques may also benefit from our work. In [39], dynamic calling contexts were used to direct static analysis to locate root causes of null references. Our technique allows collecting more contexts besides those of the exceptions. Isolating the root causes of memory leaks [40] involves propagating memory allocation points through dynamic information flow to facilitate isolating the root cause of a memory leak. The technique can be extended with context encoding to provide compact contexts of memory allocations. It was shown in [41] that the effectiveness of fault localization techniques can be affected adversely by coincidental correctness, which occurs when a fault is executed but no failure is detected. The authors propose to refine code coverage with context patterns such as control and data-flow patterns. The results show that such patterns can improve fault localization. Our technique provides the option of using calling contexts. In fact, our technique can provide context information in any runtime profile-based fault localization and debugging techniques such as [14], [42], [43], [44], [45], [46], [47], [48], [49], [50], [51], [52], [53]. Note that such information is in general very useful for programmer inspection. The benefits of context encoding extend beyond just debugging. For example, profile-based optimization and specialization [54] can acquire more fine-grained capabilty using context encoding.

12 Conclusions

We propose a technique that encodes the current calling context at any point during execution. It encodes an acyclic call path into a number and divides a recursive path into subsequences to encode them independently. It leverages stack depth to remove unnecessary encoding. The technique guarantees different contexts have different IDs and a context can be decoded from its ID. Our results show that the technique is highly efficient, with overhead of 0-6.4 percent. It is also highly effective, encoding contexts of most medium-sized programs into just one number and those of large programs in a few numbers in most cases. Our client study on tracking context sensitive definitions for debugging crash bugs shows that our approach is simple and effective.

Acknowledgments

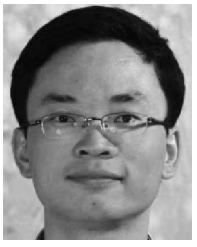
The authors thank the ICSE reviewers for their insightful comments. They are grateful to Ben Wiedermann for his feedback on early drafts. This research is supported, in part, by the US National Science Foundation (NSF) under grants 0917007, 0847900, and 0845870. Any opinions, findings, conclusions, or recommendations in this paper are those of the authors and do not necessarily reflect the views of the NSF.

References

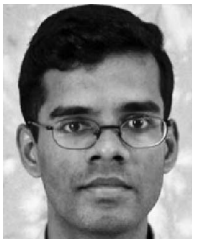
[1] X. Zhang, S. Tallam, and R. Gupta, "Dynamic Slicing Long Running Programs through Execution Fast Forwarding," *Proc. 14th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.*, 2006.
[2] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution," *Proc. 15th Ann. Network and Distributed System Security Symp.*, 2008.
[3] A. Villazon, W. Binder, and P. Moret, "Flexible Calling Context Reification for Aspect-Oriented Programming," *Proc. Eight ACM Int'l Conf. Aspect-Oriented Software Development*, 2009.
[4] Z. Lai, S.C. Cheung, and W.K. Chan, "Inter-Context Control-Flow and Data-Flow Test Adequacy Criteria for nesC Applications," *Proc. 16th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.*, 2008.
[5] G. Ammons, T. Ball, and J.R. Larus, "Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 1997.
[6] M.D. Bond and K.S. McKinley, "Probabilistic Calling Context," *Proc. 22nd Ann. ACM SIGPLAN Conf. Object-Oriented Programming Systems and Applications*, 2007.
[7] X. Zhuang, M.J. Serrano, H.W. Cain, and J.-D. Choi, "Accurate, Efficient, and Adaptive Calling Context Profiling," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 2006.
[8] R.E. Jones and C. Ryder, "A Study of Java Object Demographics," *Proc. Seventh Int'l Symp. Memory Management*, 2008.
[9] X. Zhang, A. Navabi, and S. Jagannathan, "Alchemist: A Transparent Dependence Distance Profiling Infrastructure," *Proc. IEEE/ACM Seventh Ann. Int'l Symp. Code Generation and Optimization*, 2009.
[10] M.D. Bond, G.Z. Baker, and S.Z. Guyer, "Breadcrumbs: Efficient Context Sensitivity for Dynamic Bug Detection Analyses," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 2010.
[11] T. Mytkowicz, D. Coughlin, and A. Diwan, "Inferred Call Path Profiling," *Proc. 24th ACM SIGPLAN Conf. Object Oriented Programming Systems Languages and Applications*, 2009.
[12] http://www.cs.purdue.edu/~wsumner/research/cc, 2012.
[13] T. Ball and J.R. Larus, "Efficient Path Profiling," *Proc. IEEE/ACM 29th Ann. Int'l Symp. Microarchitecture*, 1996.
[14] T.M. Chilimbi, B. Liblit, K. Mehra, A.V. Nori, and K. Vaswani, "Holmes: Effective Statistical Debugging via Efficient Path Profiling," *Proc. 31st Int'l Conf. Software Eng.*, 2009.
[15] J.R. Larus, "Whole Program Paths," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 1999.
[16] T. Reps, T. Ball, M. Das, and J. Larus, "The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem," *Proc. Sixth European Software Eng. Conf. Held Jointly with the Fifth ACM SIGSOFT Int'l Symp. Foundations of Software Eng.*, 1997.
[17] F. Pizlo, L. Ziarek, E. Blanton, P. Maj, and J. Vitek, "High-Level Programming of Embedded Hard Real-Time Devices," *Proc. Fifth European Conf. Computer Systems*, 2010.
[18] G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," *Proc. 11th Int'l Conf. Compiler Construction*, 2002.
[19] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., 1986.
[20] http://www.sventantau.de/public_files/mplayer/mplayer_20050824.txt, 2010.
[21] http://www.securityfocus.com/bid/6120, 2012.
[22] http://www.securiteam.com/unixfocus/5BP0P2A6AY.html, 2012.

[23] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas, "AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-Based Invariants," *Proc. IEEE/ACM 37th Ann. Int'l Symp. Microarchitecture,* 2004.
[24] http://www.sventantau.de/public_files/chmlib/chmlib_20051126.txt, 2010.
[25] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* 2007.
[26] J.M. Spivey, "Fast, Accurate Call Graph Profiling," *Software—Practice and Experience,* vol. 34, pp. 249-264, http://portal.acm.org/citation.cfm?id=991085.991087, Mar. 2004.
[27] A. Rountev, S. Kagan, and M. Gibas, "Static and Dynamic Analysis of Call Chains in Java," *Proc. ACM SIGSOFT Int'l Symp. Software Testing and Analysis,* 2004.
[28] D. Melski and T.W. Reps, "Interprocedural Path Profiling," *Proc. Eight Int'l Conf. Compiler Construction, Held as Part of the European Joint Conf. the Theory and Practice of Software,* 1999.
[29] J. Law and G. Rothermel, "Whole Program Path-Based Dynamic Impact Analysis," *Proc. 25th Int'l Conf. Software Eng.,* 2003.
[30] B. Wiedermann, "Know Your Place: Selectively Executing Statements Based on Context," Technical Report TR-07-38, Univ. of Texas, 2007.
[31] R. Joshi, M.D. Bond, and C. Zilles, "Targeted Path Profiling: Lower Overhead Path Profiling for Staged Dynamic Optimization Systems," *Proc. Int'l Symp. Code Generation and Optimization,* 2004.
[32] K. Vaswani, A.V. Nori, and T.M. Chilimbi, "Preferential Path Profiling: Compactly Numbering Interesting Paths," *Proc. 34th Ann. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* 2007.
[33] M. Serrano and X. Zhuang, "Building Approximate Calling Context from Partial Call Traces," *Proc. IEEE/ACM Seventh Ann. Int'l Symp. Code Generation and Optimization,* 2009.
[34] M. Arnold and D. Grove, "Collecting and Exploiting High-Accuracy Call Graph Profiles in Virtual Machines," *Proc. Int'l Symp. Code Generation and Optimization,* 2005.
[35] O. Agesen, "The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism," *Proc. Ninth European Conf. Object-Oriented Programming,* 1995.
[36] A. Milanova, A. Rountev, and B.G. Ryder, "Parameterized Object Sensitivity for Points-To Analysis for Java," *ACM Trans. Software Eng. and Methodology,* vol. 14, pp. 1-41, http://doi.acm.org/10.1145/1044834.1044835, Jan. 2005.
[37] O. Shivers, "Control-Flow Analysis of Higher-Order Languages," PhD dissertation, Carnegie Mellon Univ., May 1991.
[38] M. Sridharan and R. Bodík, "Refinement-Based Context-Sensitive Points-To Analysis for Java," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* 2006.
[39] S. Sinha, H. Shah, C. Görg, S. Jiang, M. Kim, and M.J. Harrold, "Fault Localization and Repair for Java Runtime Exceptions," *Proc. 18th Int'l Symp. Software Testing and Analysis,* 2009.
[40] J. Clause and A. Orso, "Leakpoint: Pinpointing the Causes of Memory Leaks," *Proc. IEEE/ACM 32nd Int'l Conf. Software Eng.,* 2010.
[41] X. Wang, S.C. Cheung, W.K. Chan, and Z. Zhang, "Taming Coincidental Correctness: Coverage Refinement with Context Patterns to Improve Fault Localization," *Proc. 31st Int'l Conf. Software Eng.,* 2009.
[42] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Practical Fault Localization for Dynamic Web Applications," *Proc. IEEE/ACM 32nd Int'l Conf. Software Eng.,* 2010.
[43] E.D. Berger and B.G. Zorn, "Diehard: Probabilistic Memory Safety for Unsafe Languages," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* 2006.
[44] M.D. Bond, N. Nethercote, S.W. Kent, S.Z. Guyer, and K.S. McKinley, "Tracking Bad Apples: Reporting the Origin of Null and Undefined Value Errors," *Proc. 22nd Ann. ACM SIGPLAN Conf. Object-Oriented Programming Systems and Applications,* 2007.
[45] Y. Brun and M.D. Ernst, "Finding Latent Code Errors via Machine Learning over Program Executions," *Proc. 26th Int'l Conf. Software Eng.,* 2004.
[46] O.C. Chesley, X. Ren, B.G. Ryder, and F. Tip, "Crisp—A Fault Localization Tool for Java Programs," *Proc. 29th Int'l Conf. Software Eng.,* 2007.
[47] T.M. Chilimbi and V. Ganapathy, "HeapMD: Identifying Heap-Based Bugs Using Anomaly Detection," *Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* 2006.
[48] P. Francis, D. Leon, M. Minch, and A. Podgurski, "Tree-Based Methods for Classifying Software Failures," *Proc. 15th Int'l Symp. Software Reliability Eng.,* 2004.
[49] J.A. Jones and M.J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique," *Proc. IEEE/ACM 20th Int'l Conf. Automated Software Eng.,* 2005.
[50] A.J. Ko and B.A. Myers, "Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior," *Proc. 30th Int'l Conf. Software Eng.,* 2008.
[51] B. Liblit, M. Naik, A.X. Zheng, A. Aiken, and M.I. Jordan, "Scalable Statistical Bug Isolation," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* 2005.
[52] C. Liu, X. Yan, L. Fei, J. Han, and S.P. Midkiff, "Sober: Statistical Model-Based Bug Localization," *Proc. 10th European Software Eng. Conf. held jointly with 13th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* 2005.
[53] S. Park, R.W. Vuduc, and M.J. Harrold, "Falcon: Fault Localization in Concurrent Programs," *Proc. ACM/IEEE 32nd Int'l Conf. Software Eng.,* 2010.
[54] M. Arnold, M. Hind, and B.G. Ryder, "Online Feedback-Directed Optimization of Java," *Proc. 17th ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications,* 2002.

William N. Sumner received the BS degree in computer science and mathematics from Hope College, Michigan. Currently, he is working toward the PhD degree in the Department of Computer Science at Purdue University. His interests include dynamic analysis, automated debugging, and bug detection for sequential, and concurrent programs.



Yunhui Zheng received the BE degree in computer science from the University of Science and Technology, China. Currently, he is working toward the PhD degree in the Department of Computer Science at Purdue University. He is interested in dynamic program analysis, web applications, and how they relate to software security.



Dasarath Weeratunge received the BSc (Eng.) degree in computer science and engineering from the University of Moratuwa, Sri Lanka. Currently, he is working toward the graduate degree in the Department of Computer Science at Purdue University. His research interests include dynamic program analysis and debugging concurrent programs.

Xiangyu Zhang received the BS and MS degrees from the University of Science and Technology, China, and the PhD degree from the University of Arizona in October 2006. Currently, he is working as an assistant professor at Purdue University. His research interests lie in techniques on automatic debugging, dynamic program analysis, software reliability, and security. He is the recipient of the 2006 ACM SIGPLAN Dissertation Award. He is a member of the IEEE.