

Data-Delineation in Software Binaries and its Application to Buffer-Overflow Discovery [preprint]

Denis Gopan*, Evan Driscoll*, Ducson Nguyen[†], Dimitri Naydich[†], Alexey Loginov[†] and David Melski*
GrammaTech, Inc.

*Madison, WI, USA [†]Ithaca, NY, USA

{gopan,edriscoll,dnguyen,dnaydich,alexey,melski}@grammatech.com

Abstract—Detecting memory-safety violations in binaries is complicated by the lack of knowledge of the intended data layout, i.e., the locations and sizes of objects. We present lightweight, static, heuristic analyses for recovering the intended layout of data in a stripped binary. Comparison against DWARF debugging information shows high precision and recall rates for inferring source-level object boundaries. On a collection of benchmarks, our analysis eliminates a third to a half of incorrect object boundaries identified by an IDA Pro-inspired heuristic, while retaining nearly all valid object boundaries.

In addition to measuring their accuracy directly, we evaluate the effect of using the recovered data for improving the precision of static buffer-overflow detection in the defect-detection tool CodeSonar/x86. We demonstrate that CodeSonar’s false-positive rate drops by about 80% across our internal evaluation suite for the tool, while our approximation of CodeSonar’s recall only degrades about 25%.

I. INTRODUCTION

Despite an orchestrated effort by the research community, buffer overruns still constitute one of the most serious threats to software security. Many techniques for protecting *arbitrary programs* have been advanced, for example, stack canaries [1], ASLR [2, 3], DEP [4], shadow stacks [5, 6], and control-flow integrity [7–10]. These techniques provide coarse-grained protections that are effective against attacks that attempt to hijack the control flow of a program.

However, some attacks are much harder to detect and remediate: non-control data attacks modify critical data without directly affecting control flow [11], and buffer *overreads* can expose sensitive data, again without overt effects on the program’s execution. Against these kinds of attacks, the techniques mentioned in the previous paragraph offer little to no protection.

If the positions and sizes of buffers in a program are known, then more powerful protection techniques can be applied, for example fine-grained stack-layout transformation [12] or insertion of explicit memory-safety checks [13–16]. In addition, static analysis can be applied to find potential vulnerabilities. Unfortunately, this information is not readily available for stripped binaries.

This material is based upon work supported by the United States Air Force under Contract No. FA8650-10-C-7025. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force.

Existing techniques for inferring object boundaries tend to fall short in one of two ways. Some heuristic-based approaches, e.g., IDA Pro [17], often break large objects into pieces. This can result in false alarms if used for bounds checking, and may even disrupt program functionality if used for program transformation. Other approaches, e.g., [18, 19], assume that a program is memory safe, and thus derive bounds that include the potential overruns. The resulting information is unhelpful for buffer-overflow detection.

In this paper, we present a heuristic approach to the following:

Given an arbitrary stripped executable, infer locations and sizes of objects suitable for buffer-overflow detection and protection.

We call our approach *Data-Delineation Analysis* (DDA). We use the term *object* to refer to any top-level datum, such as an array, structure, or variable, regardless of whether or not the binary was created from an object-oriented language. DDA first finds a set of object boundaries that is largely a superset of the desired result, and then it systematically refines this set by eliminating boundaries that fall within larger objects, such as structure instances. Insofar as the initial set of boundaries provides an overapproximation to the ground truth, our goal is to eliminate as many boundaries that cause false-positive buffer-overflow warnings as we can; at the same time, we want to retain as many boundaries that enable overrun detection as possible.

For the second step of DDA, we use a novel technique that we call *Parameter-Offset Analysis* (POA). POA operates by identifying and symbolically propagating possible “base” pointers to objects. For each base pointer, the set of constant offsets used in memory dereferences is collected to estimate the extent of the object pointed to by the base pointer. That is, if the analysis sees an instruction “`mov eax, [ebx + 128]`” and the symbolic value of `ebx` is base pointer P_{base} , the analysis infers that the instruction is performing a field access and concludes that the object pointed to by P_{base} is at least 132 bytes long. (The memory access above reads four bytes.)

To avoid the pitfall of over-approximating object extent based on unsafe memory accesses, we rely on the intuition that buffer overruns are more likely to be in loops where the address or offset changes each iteration. Therefore, our analysis

only makes use of offsets that appear as syntactic *constants* in the program text.

We implemented our approach for 32-bit x86 binaries and evaluated the analysis prototype on a number of real-world benchmarks – binaries compiled from C and C++ code with both GNU GCC and the Microsoft Visual C++ compiler. To evaluate our analysis we conducted two types of experiments:

- We compared the boundaries inferred by our analysis with debug information (DWARF2). The evaluation showed that our approach yields a low number of both false positives and false negatives for unoptimized builds. However, the analysis precision somewhat decreases at more aggressive optimization levels. We present a detailed analysis of false positives and false negatives in our experiments.
- We used object boundaries derived by our analysis to improve precision of CodeSonar/x86 [20], a commercial heuristic static-analysis tool for detecting defects in x86 binaries. Our experiments showed that the better object boundaries found using parameter-offset analysis reduced our approximation of the number of false-positive warnings by about 80% while only reducing our approximation of the number of true positives by slightly more than 25% (which corresponds to increasing false negatives). Interestingly, optimized programs saw *more* benefit from DDA on both measures.

Contributions. This paper makes the following contributions:

- We present an approach for recovering data layout in an arbitrary, stripped binary.
- We describe a scalable implementation of our approach
- We present an extensive evaluation of our prototype implementation based on comparison of the recovered layout data with debug information.
- We evaluate the usefulness of the improved data boundaries when used for static buffer-overflow detection, using CodeSonar/x86.

II. OVERVIEW

Figure 1 shows a selection of code that is simplified from the SSH client PuTTY [21]. When CodeSonar/x86 is in a configuration that allows it to find object overruns (see “Evaluation Methodology” in Sect. V) but without running POA, the tool reports a false-positive buffer overrun on Lines 20 and 21. The root cause of this overrun is poor object-boundary identification. In this section, we will explain why this warning surfaces, and then examine the operation of POA through the example, showing how it improves the object boundaries and eliminates the warning.

Motivation Without POA, CodeSonar/x86 uses object boundaries largely as they are found by IDA Pro. IDA Pro determines what boundaries exist based in part on the addresses that are explicitly used in an instruction.

In Fig. 1, because addresses `esp`, `esp+128`, and `esp+132` appear explicitly within `des3_decrypt_pubkey_ossh`, IDA Pro inserts object boundaries at those locations and

CodeSonar/x86 uses those boundaries in its analysis. However, if we look at the source, we see that `var_118` at `esp+128` (corresponding to `ourkeys.iv0`) and `var_114` at `esp+132` (`ourkeys.iv1`) are in the middle of the `ourkeys` object, thus `ourkeys` is split up in IDA Pro and hence in CodeSonar/x86’s intermediate representation (IR).¹ What this means is that the portion of the object corresponding to the `k0246` and `k1357` fields is smaller than it should be – only 128 bytes instead of the actual struct size of 136 bytes.

When CodeSonar/x86 analyzes the program for buffer overruns, it sees the path from `des3_decrypt_pubkey_ossh` into `des_cbc3_decrypt` and examines what happens along that path. What it sees concerns it: CodeSonar determines that Line 20 is accessing offsets 128–131 of `var_198` and Line 21 is accessing offsets 132–135 within that object, but it thinks that `var_198` is only 128 bytes. As a result, it reports buffer-overrun warnings for both lines.

Parameter-Offset Analysis What we call POA in this paper is a collection of related heuristics; we will now describe the operation of the most fruitful of these (and the origin of the name). Sect. III includes some discussion of the others.

Accesses to fields of structured data are often compiled into instructions that add a constant offset to a base pointer and then dereference it, e.g., those instructions in Lines 20 and 21 of Fig. 1. In that example, the base pointer is passed in as the parameter to `des_cbc3_decrypt` and is then loaded into `eax` on Line 19. Syntactic-constant offsets 128 and 132 from that base pointer are then computed and dereferenced.

Meanwhile, the most likely buffer overruns occur either in calls to library functions (e.g., the infamous `gets`) or inside loops; neither one of these constructs causes access patterns with syntactic constants.

These two observations led us to our guiding principle:

Syntactic constants have a good chance of being correct.

Returning to our running example, our position is that the offsets accessed on Lines 20 and 21, because they are constants, are much more likely to be correct than incorrect. As a result, POA concludes “`var_198` (the base pointer for those accesses) is at least 136 bytes” and removes the boundaries that IDA Pro inserted at 128 and 132 bytes.

POA works bottom-up over the call graph, computing a summary for each procedure. The summary, which we call a model, consists of a map from a parameter ordinal to a set of offsets that are added to the parameter in an address computation and then dereferenced. At call sites, POA looks at the model for the called procedure, looks at each argument passed in the call, and if there is a non-empty set associated with that parameter, then POA will treat the value being passed as a base pointer and remove any boundaries that fall within the indicated base-offset range. The final step is contingent on the value of the argument being the address of an object in a form that POA recognizes; see “Offset Collection and Propagation” within Sect. III.

¹The NNN in each `var_NNN` is the hex offset from the base of the frame.

```

1 typedef struct {
2     word32 k0246[16], k1357[16];
3     word32 iv0; // offset 128
4     word32 iv1; // offset 132
5 } DESContext;

6 void des3_decrypt_pubkey_ossh()    sub_427270:
7 {                                sub esp, 408
8     DESContext ourkeys;
9     ourkeys.iv0 = ...;            mov dword [esp + 132], ...
10    ourkeys.iv1 = ...;            mov dword [esp + 128], ...
11    des_cbc3_decrypt(&ourkeys);    lea ecx, [esp]
12    ...                            push ecx
13    ...                            call sub_426C10
14 }

15 static void des_cbc3_decrypt(
16     DESContext * scheds)          sub_426C10:
17 {
18     word32 iv0, iv1;              ...
19     ....                          mov eax, dword [ebp + 8]
20     scheds->iv0 = iv0;            mov dword [eax + 128], ebx
21     scheds->iv1 = iv1;            mov dword [eax + 132], edx
22 }

```

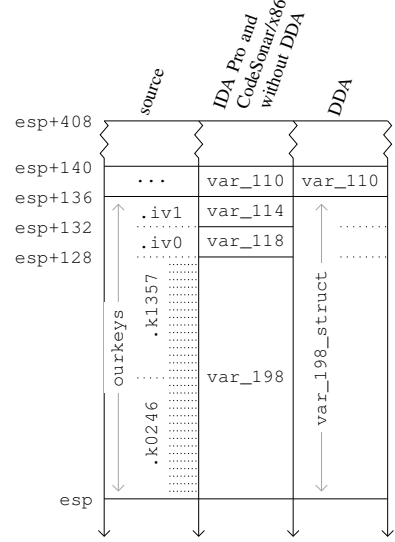


Fig. 1: Running example, simplified from PuTTY. Shows source excerpts, the corresponding machine code, and a diagram of the stack under different views. The dotted lines in the source diagram indicate subobject boundaries. The dotted lines in the DDA diagram also indicate subobject boundaries, but these boundaries are not currently used by CodeSonar/x86 and we do not evaluate them in this paper (only top-level objects). The vertical axis is not to scale; each element of `ourkeys.k0246` and `ourkeys.k1357` is four bytes as well. The object `var_110` is included because of an access to `esp+110` that is omitted from the excerpt.

For Fig. 1, POA first examines `des_cbc3_decrypt`. Using symbolic execution, it tracks the flow of the first parameter (represented by a symbolic constant p_1) into `eax` on Line 19. On Line 20, the analysis sees a memory dereference with the symbolic address $p_1 + 128$. The access is four bytes, and so POA adds to `des_cbc3_decrypt`'s model the mapping $\{p_1 \mapsto \{128, \dots, 131\}\}$. The analysis acts similarly for Line 21, leading to the overall model $\{p_1 \mapsto \{128, \dots, 135\}\}$.

Next, POA examines `des3_decrypt_public_ossh`. At the call site at line 13, the analysis sees that there is a non-trivial set for p_1 , and thus looks at what is being passed as the first parameter; in this case, the analysis will have computed the value of the first parameter as follows. The assembly instruction on Line 11 loads `esp` (`&var_198`) into the symbolic state for `ecx`. Line 12 propagates that address from `ecx` to the parameter space on the stack, which is not modified before the function call. As a result, parameter-offset analysis sees that `&var_198` is passed as the first parameter. It combines that with the model information to conclude that `var_198` is at least 136 bytes.

As the final step (separate from POA proper), the IR used by CodeSonar/x86 is modified to merge the three objects at `esp`, `esp+128`, and `esp+132` according to this new size information (thus creating a 136-byte structure). This corrects the data-boundary information and eliminates the reason for the warning, which no longer appears when running with parameter-offset analysis. Note that CodeSonar/x86 currently only looks at overruns of an entire top-level object: it does not try to find overruns of a field, so the fact that the resulting

object in the IR is a structure instead of a monolithic object is currently ignored.

At this point DDA is complete, and the resulting objects are ready to be consumed by an end analysis, e.g. CodeSonar/x86's warning searches.

III. DATA-DELINEATION ANALYSIS

Data-delineation analysis is comprised of three principal steps:

- Step 1:** Heuristically extract a superset of potential objects, S_{sup} . DDA finds S_{sup} by looking at offsets from the frame or stack pointer.
- Step 2 (POA):** Identify a set of aggregate objects, S_{agg} . DDA finds S_{agg} by looking at offsets from the base pointers of objects in S_{sup} .
- Step 3:** Construct the resulting list of objects by extending S_{agg} with those objects in S_{sup} that are not subsumed by any of the aggregate objects.

The main contribution of this paper is the analysis for performing Step 2 above. Steps 1 and 3 are likely to be specific to the clients of the overall data-delineation analysis.

A. Initial Object Identification

As we mentioned above, the identification of the initial set of objects is likely to be tied to the implementation specifics of a DDA client. For instance, many binary analysis tools use commercial IDA Pro disassembler for initial processing of an executable and may adopt the sets of objects identified by IDA Pro as S_{sup} . This is perfectly acceptable to our analysis –

in fact, CodeSonar uses of IDA Pro along with several other sources of information to compute its S_{sup} .

In addition to CodeSonar/x86, we have a standalone implementation of DDA. For this implementation, we implemented a heuristic that we believe captures the spirit of IDA Pro’s object identification; we call this the *Constant-Offsets Heuristic*. To build a list of local objects, our implementation scans each function’s instructions looking for dereferences of the form $reg + c$, where reg is the stack or frame pointer (`esp` or `ebp` on x86) and c is a constant. For each such access, we add an object with the starting offset derived from c . We do not explicitly derive sizes for the identified objects; instead, we assume that each object extends until the next identified object.

B. Parameter-Offset Analysis

Parameter-Offset Analysis (POA) is the main contribution of this paper. The analysis operates by identifying base addresses of aggregate objects and collecting sets of constant offsets that are used in memory dereferences in conjunction with each base address. The collected sets of offsets are used to estimate objects sizes. The subsequent three sections describe how the analysis collects and propagates the offsets, how the base addresses are recognized, and how the resulting set of aggregate objects is derived.

1) *Offset Collection and Propagation*: The analysis constructs a model of each function in a program. The models of callees are used to build the model of the caller. Let f be a function. A model of f constructed by the analysis is a map $M_f : P \rightarrow \wp(Z)$, where P is the set of f ’s parameters and Z is the set of integers. The model maps each parameter to the set of constant offsets that are used in dereferences of the parameter’s value. Consider function `des_cbc3_decrypt` in Figure 1: the model that the analysis constructs for this function has the form $\{p_1 \mapsto \{128, \dots, 135\}\}$. This captures the fact that the first parameter of the function is used as a base address with offsets 128–135 dereferenced.

To find instructions that dereference pointers passed in parameters, the analysis uses intraprocedural symbolic execution. The symbolic execution starts at the function entry point with a symbolic state in which parameter locations are initialized with symbolic constants. (In x86 code, the parameters are either placed on the stack above the return address or passed in general-purpose registers.) For expediency, we perform symbolic execution of all paths in a spanning tree of each function’s CFG. This approximation greatly speeds up the analysis, and we have only seen a few cases in which it caused analysis to be imprecise.

We recognize four major sources of offset information:

Explicit dereferences. We check the symbolic value of the dereferenced memory address. If the address has the form $p_i + c$, where p_i is a symbolic constant that corresponds to a parameter and c is a constant value (which may vacuously be zero) and where k is the size of the access, the model of the function is extended as follows:

$$M_f = M_f[p_i \mapsto M_f(p_i) \cup \{c, c + 1, \dots, c + k - 1\}]$$

Function calls. A parameter may be dereferenced by a callee of a function if it is passed transitively. At each call site, we check the symbolic values of each parameter to be passed to the callee. If the value of callee parameter j is of the form $p_i + c$, the model is updated as follows:

$$M_f = M_f[p_i \mapsto M_f(p_i) \cup \text{translate}(M_g(p_j), c)]$$

where $\text{translate}(S, c) = \{s + c \mid s \in S\}$ and M_g is the model of the callee.

Buffer-manipulation library calls. Calls to standard buffer and string manipulation functions (e.g., `memset` and `strcpy`) may dereference parameter values. These functions typically take a buffer base pointer and length as parameters. Often the length is a syntactic constant generated by a `sizeof` operation in the source, which we will like to consider trustworthy. As a result, at a call site of such a function, we inspect the symbolic values of these parameters. If the buffer address has the form $p_i + c$ and the length parameter has a constant value l , we extend the model as follows:

$$M_f = M_f[p_i \mapsto M_f(p_i) \cup \{c, c + 1, \dots, c + l - 1\}]$$

Hardware-assisted loop instructions. The x86 instruction set contains several instructions that are executed repeatedly in a hardware-induced loop until a certain condition is met. For instance, “`rep stos`” writes the value from register `eax` to the memory word addressed by the destination register `edi`, decrements the counter register `ecx`, updates the address in `edi` based on the direction flag `df`, and repeats until the value in the counter register becomes zero. For each such instruction, we inspect the symbolic values of source and destination registers, counter register, and direction flag. If the counter register and direction flag have constant values k and d respectively, and source or destination register has the symbolic value $p_i + c$, we update the model as follows:

$$M_f = M_f \left[p_i \mapsto M_f(p_i) \cup \left\{ c + i \mid \begin{array}{l} \text{abs}(i) < k \\ \wedge \\ \text{sign}(i) = d \end{array} \right\} \right]$$

The analysis must have the models of the callees to build the model of the caller. This is not possible for functions that are called recursively. To deal with this issue, our implementation of the analysis removes back edges from the call graph and topologically sorts the resulting acyclic graph to determine the order in which the functions must be analyzed. Again, we have not observed any significant effects of this implementation choice on the precision of the analysis.

2) *Identifying Base-Address Construction Points*: The goal of this analysis phase is to map base addresses of aggregate objects to sets of offsets that are used in dereferences of those base objects. For each function and for the global data region, the analysis computes a map with signature $Z \rightarrow \wp(Z)$. We will use L_f to denote the map for function f and denote the global map as L_G .

This analysis phase is almost identical to the offset-collection analysis, except that instead of symbolic parameter constants we look for symbolic global and local addresses. For instance, at each call site, we inspect the symbolic values of arguments that are being passed to the callee g : if the value of parameter j is an address and $M_g(p_j)$ is a non-empty set, we update the maps as follows:

The parameter value is a global address, A . The global map is updated:

$$L_G = L_G [A \mapsto L_G(A) \cup M_g(p_j)]$$

The parameter value is a local address, $sp + C$, where sp is the symbolic value of the stack pointer at function entry. We update the map for the analyzed function as follows:

$$L_f = L_f [C \mapsto L_f(C) \cup M_g(p_j)]$$

We treat the calls to buffer-manipulation functions and the invocations of hardware-assisted loop instructions similarly. The explicit memory dereferences, however, are skipped – we rely on the constant-offsets heuristic from Sect. III-A to recover the corresponding objects.

In our implementation, we simplified the above approach slightly: we seeded the symbolic execution with a fixed initial value for the stack pointer. This speeds up the analysis as it opens up more opportunities for constant folding during symbolic execution, and because addresses are always a constant integer, it simplifies handling. As in the previous approximation, no ill effects have been observed as a result.

Local-Offset Analysis. DDA can get additional information from an intra-procedural counterpart to POA. If a base address is loaded into a register locally, an offset computed, and then a dereference occurs, we make a corresponding inference. The following example illustrates a place of potential benefit:

```
struct S s, *p;
p = &s;          1: lea  eax, [ebp-0x28]
p->x = 5;         2: mov  DWORD PTR [eax+0x20], 0x5
```

To account for such cases, we extended the analysis as follows: if an instruction assigns an address of a local variable (i.e., a symbolic value $sp+C$) to a register, we replace the value of that register with specially constructed symbolic constant l_C . At the places where offset sets are collected (memory dereferences, function calls, library calls, and hardware-assisted loops), we check if the dereferenced address has the form $l_C + d$, and if so, we update the L_f mapping for C accordingly.

3) *Aggregate Object Extraction.*: The final step in our analysis is to extract the actual objects from the L -maps described above. The process we follow is trivial: for each mapping in the L map, we extract the set of offsets and determine the starting address of the object (note that offsets in the set may potentially be negative) and its extent. The objects that overlap are merged together. All of the created objects are added to set S_{agg} .

IV. DDA ACCURACY EVALUATION

In this section, we describe our evaluation of the accuracy of the boundaries inferred by DDA. We applied our standalone

implementation of the analysis to a number of real-world 32-bit x86 programs. Table I presents the analysis results. The first block in the table shows the results for Coreutils binaries (100 small utilities for basic file, shell, and text manipulation). We analyzed the Coreutils suite built at various optimization levels to investigate the effect the optimization has on DDA's results. The next two blocks of benchmarks in the table are larger C programs and C++ programs, respectively. These were compiled without optimization. All benchmarks were built on Ubuntu 12.04 with GCC 4.6.3.

Evaluation Methodology. We compared the set of inferred objects to ground truth as defined by the DWARF2 debug information emitted by GCC. In this set of experiments, we only concerned ourselves with local variables, ignoring globals. In addition, we only compare the parts of each activation record that correspond to explicit variables in the source. (Function frames typically store additional data, such as compiler-introduced temporaries and spilled register values, that DDA will infer information about, but which has no associated debug information.)

There are two types of imprecision our analysis yields. False positives correspond to inferred object boundaries that do not have counterparts in the ground truth. False negatives correspond to ground-truth object boundaries that are missed by the analysis. We measure both of these:

Precision: is a measure of how many spurious boundaries the analysis infers (i.e., a measure of false positive rate).

Recall: is a measure of how many true objects the analysis misses (i.e., the measure of false negative rate).

We use the following formulas to compute the two metrics. Let M denote the number of object boundaries that the analysis correctly identified, FP denote the number of false positives, and FN denote the number of false negatives. Also, let $GT = M + FN$ denote the number of ground-truth object boundaries.

$$Precision = \frac{GT}{GT + FP} \times 100\% \quad Recall = \frac{M}{GT} \times 100\%$$

The two metrics must be used together to assess the accuracy of the analysis: for an accurate analysis, both precision and recall should be close to 100%. In separation, the metrics could be easily misinterpreted: e.g., finding no boundaries yields 100% precision, but 0% recall; similarly, inferring that each byte in the activation record is a separate object, yields 100% recall, but low precision.

Analysis Results. Tab. I presents the experimental evaluation of our analysis. We report both precision (columns on the left) and recall (columns on the right) for several different analysis configurations. CO shows the results for the Constant Offsets heuristic from Sect. III-A (i.e., when no POA is used). This data represents a good baseline for the evaluation of the analysis: it shows the lowest obtainable precision and the highest obtainable recall for boundary identification. We also show the results obtained by using standard IDA Pro mechanisms for detecting local variables (column IDA in the table). Note that IDA

Benchmark	Vars	PRECISION (%)							RECALL (%)						
		IDA	CO	DEREF	+LOCAL	+LIB	ALL	+64BIT	IDA	CO	DEREF	+LOCAL	+LIB	ALL	+64BIT
Coreutils	214	81.64	80.99	87.40	87.42	88.17	88.23	94.36	95.00	98.94	98.94	98.93	98.91	98.91	97.66
Coreutils (-O)	76	59.03	56.41	72.66	72.66	74.32	74.45	85.37	77.23	86.01	85.96	85.97	85.86	85.86	84.29
Coreutils (-O2)	86	38.98	37.86	54.75	54.75	55.72	55.80	77.96	74.79	82.08	80.57	80.57	80.48	80.48	80.16
grep	1043	86.63	86.48	92.55	92.55	93.29	94.22	96.93	98.37	99.81	99.81	99.81	99.71	99.71	99.33
irssi	4635	98.16	97.99	98.49	98.49	99.02	99.02	99.34	97.78	97.78	97.78	97.78	97.73	97.73	96.63
nginx	1056	52.35	50.99	77.59	77.59	80.92	86.35	91.43	99.62	99.91	99.91	99.91	99.91	99.91	99.05
tcpdump	588	59.33	57.53	67.82	67.82	69.10	72.86	80.33	96.43	99.83	99.83	99.83	99.66	99.66	97.96
wget	1954	88.02	85.36	87.66	87.66	91.44	91.91	95.78	99.90	99.85	99.85	99.85	99.64	99.64	98.41
zsh	3884	91.32	90.43	91.67	91.67	94.41	94.66	96.86	99.95	99.95	99.92	99.92	99.79	99.79	98.56
average	2193	79.30	78.13	85.96	85.96	88.03	89.84	93.45	98.68	99.52	99.52	99.51	99.41	99.41	98.32
llvm-as	8145	-	95.27	96.21	96.21	96.36	96.36	-	-	99.90	99.90	99.90	99.90	99.90	-
llvm-diff	7189	-	94.23	95.43	95.43	95.55	95.55	-	-	99.83	99.83	99.83	99.83	99.83	-
llvm-mc	11937	-	84.46	88.03	88.04	88.17	88.17	-	-	99.61	99.61	99.61	99.61	99.61	-
llvm-objdump	13682	-	83.82	87.33	87.49	87.87	87.87	-	-	99.60	99.60	99.60	99.60	99.60	-
llvm-size	1721	-	84.14	90.10	90.10	92.93	92.93	-	-	99.88	99.88	99.88	99.88	99.88	-
doxygen	14351	-	98.15	98.61	98.61	98.80	98.80	-	-	99.98	99.98	99.97	99.97	99.97	-
average	9504	-	90.03	92.62	92.65	93.28	93.28	-	-	99.80	99.80	99.80	99.80	99.80	-

TABLE I: Analysis evaluation. The “Vars” column shows the number of local variables in the ground truth data. Column IDA shows the measurements for IDA Pro object detection. Column CO shows measurements for the Constant Offsets heuristic. The remaining columns show how those measurements are improved with the use of additional POA heuristics. Column DEREF shows what happens if aggregate objects are identified based on explicit dereferences only. Column +LOCAL shows how measurements improve if the Local Offset extension is enabled. Column +LIB shows the effect of taking buffer-manipulation functions into consideration. Column ALL shows the measurement for the full-scale data-delineation analysis. Finally, column +64BIT shows the DDA results are affected by a simple heuristic for detection of 64-bit integers.

Pro results are generally comparable to those of constant-offset heuristic. IDA Pro sports slightly higher precision than constant-offset heuristic, which is due to IDA Pro’s use of FLIRT for recognizing library calls and using that information to infer types for some of the variables. Surprisingly, IDA Pro’s recall is slightly lower. Our investigation showed that it was caused by IDA Pro’s failure to recognize variables in function `main(...)` for some of the benchmarks, possibly due to the presence of stack-alignment operations. Due to time constraints we were unable to collect IDA Pro data for the C++ benchmarks.

The subsequent columns show how the results are affected by using various configurations of POA. DEREF shows the measurements for a variant of POA that relies only on explicit memory dereferences. +LOCAL shows the measurements for explicit-dereferences-only POA with the Local Offsets Analysis extension enabled. +LIB shows measurements if calls to buffer-manipulation functions are also taken into consideration. Finally, ALL shows the measurements for the complete DDA.

As the data in Tab. I demonstrates, compared to the Constant Offsets heuristic, our analysis improves the precision of data delineation by 12% for C programs (by 10% with respect to IDA Pro) and by 3.2% for C++ programs without significantly affecting the recall. That is, the use of POA eliminates about half of the false positives yielded by the Constant-Offsets heuristic for C programs, and about third of those for the C++ programs.

The experiments with the Coreutils show that optimization significantly decreases the precision of the analysis (though, the gain compared to the constant-offsets heuristic is increased).

Our investigation showed that the primary culprit is function inlining: the instances of structs are created and manipulated locally, and thus POA is not able to identify them.

We conducted an extensive study of the false positives and false negatives that our analysis yields and identified several causes:

64-bit ints: In a 32-bit build, GCC uses two 32-bit words that are manipulated separately to implement 64-bit integer types. Thus, our analysis is not able to connect the two words. This category accounts for the majority of false positives we saw in our benchmarks. To estimate how much this category of false positives affects the analysis precision, we implemented a simple heuristic for detecting 64-bit integers. The heuristic identifies pairs of adjacent words in each stack frame that are always read or written together within a basic block and marks them as 64-bit words. The column +64BIT shows how applying this simple heuristic affects the overall results of DDA. On average, the heuristic removes about half of the remaining false positives. However, it also has an effect (sometimes sizable) on the analysis recall. Additional work is needed to make 64-bit integer detection more precise.

Incomplete IR: The standalone DDA implementation’s reconstruction of program’s intermediate representation is not yet complete and does not always recover the entire control flow of a program. In particular, indirect control transfers pose a challenge: indirect function calls and control flow due to jump tables is not always properly resolved. This causes both false negatives and false positives.

Benchmark	#	TP PROXY (higher is better)			FP PROXY (lower is better)		
		DEFAULT		DDA	DEFAULT		DDA
		F.O.	OO.	OO.	F.O.	OO.	OO.
Windows Dbg. C	29	0.6	0.9	0.6	0.2	4.8	1.2
Windows Dbg. C++	3	0.0	0.0	0.0	0.3	5.3	2.3
Windows Debug	32	0.6	1.0	0.6	0.5	10.1	3.5
Windows Rls. C	29	0.4	0.7	0.6	0.3	6.3	1.9
Windows Rls. C++	3	0.0	0.0	0.0	0.2	13.6	0.9
Windows Release	32	0.4	0.7	0.6	0.5	19.9	2.9
All Windows	64	0.9	1.7	1.2	1.0	30.1	6.4
Linux	4	0.0	0.1	0.1	0.0	0.2	0.1
All	68	1.0	1.8	1.3	1.0	30.3	6.5

TABLE II: CodeSonar/x86 warning summary information. F.O. stands for FRAMES ONLY and OO. for OBJECT OVERRUN. The column # gives the number of binaries measured. The data shown here are normalized to the total number of DEFAULT BOUNDARIES/OBJECT OVERRUN warnings in the respective category (matched or binary-only).

Passing structures by value: Our techniques are geared towards programs that pass aggregate objects by reference and yields false positives when they are passed by value.

Path insensitivity: We have seen several examples in which POA over-estimated the size of an object because it ignored conditions under which specific paths can be executed. Over-estimation of object size causes the overall analysis to yield false negatives.

Accesses to explicit buffer elements: If a buffer is used locally and its elements are explicitly accessed by the program (e.g., `int rgb[3]; green = rgb[1];`), our analysis will not be able to identify it.

These shortcomings of the analysis remain to be addressed in our future work.

V. STATIC BUFFER-OVERRUN DETECTION EVALUATION

In this section, we present our evaluation of the effect of DDA on CodeSonar/x86’s ability to find buffer overruns. We first present an brief overview of our heuristic static-analysis tool CodeSonar/x86. We then describe our evaluation methodology, followed by the results.

CodeSonar Overview. CodeSonar is GrammaTech’s commercial tool that specializes in static defect detection [20]. CodeSonar was originally developed for C and C++ programs, but recently we have been retargeting it to also work on machine code. Regardless of the language, CodeSonar uses heuristic analyses to find a diverse set of defects including memory-safety errors and API misuses.

The machine-code version of CodeSonar begins with front-end analyses that build an IR of the program. It uses IDA Pro for disassembly and initial IR inference, but layers many additional advanced analyses to increase the precision of the recovered IR. The DDA implementation resides in the front end because the IR must contain information about the objects present in a program.

The CodeSonar back end then analyzes the IR to determine potential defects. The IR that the back end uses is common between the different source and binary front ends. The back end’s analyses are heuristics that scale to very large, industrial-sized programs.

Evaluation Methodology. As a portion of CodeSonar/x86’s routine testing, we analyze several dozen open-source programs to watch for unexpected changes in the reported warnings.

The number of programs and number of warnings are large enough that we cannot manually classify each warning to determine whether it is a true positive or a false positive. Instead, we use a proxy for these measurements.

The evaluation procedure for CodeSonar/x86’s warnings proceeds as follows:

- 1) We start with the source version of a program, and build it to create a binary.
- 2) We run CodeSonar/C/C++ on the source version of the program to produce a list of source warnings.
- 3) We run CodeSonar/x86 on the binary to produce a list of binary warnings.
- 4) For each binary warning produced by Step 3, we (automatically) classify it into one of three categories:

Matched Warnings: The binary warning appears to be the same as some source warning; these warnings are our proxy for the true-positive warnings in the binary analysis.

Binary-Only Warnings: The binary warning is not the same as any source warning, but is warning about a location in the program’s source; these warnings are our proxy for the false positives.

Out-Of-Scope Warnings: The binary warning was in an external library (e.g., the C runtime) rather than application code; these warnings are ignored for purposes of estimating warning accuracy.

Finding correspondences between warnings (Step 4) is done by using debugging information for the binary to get the source file and line number of each binary warning, then looking for a source warning of the same type on the same source line. Note that the use of debugging information in this process is only during the evaluation of the results and not during analysis.

We use this measure because CodeSonar/C/C++ is a leading static-analysis tool, and thus we trust the warnings *it* produces to be a reasonable approximation to ground truth. If CodeSonar/x86 finds a warning that CodeSonar/C/C++ did not find or vice versa, we expect the chance that it is caused by imprecise IR recovery (relative to the source) to be much higher than the chance that the binary analysis *actually* did a better job – though this is by no means guaranteed.² Out-of-scope warnings correspond to portions of the binary for which there was no source and thus the source analysis had no opportunity find

²Indeed, one motivation behind producing a binary version of CodeSonar is to make it possible to find “WYSINWYX” (What You See Is Not What You eXecute) problems [23], where defects and vulnerabilities only arise because of compiler transformations or low-level implementation decisions.

the warnings; we believe excluding them results in a more apples-to-apples comparison.

CodeSonar/x86 supports two modes of operation for finding buffer overruns. The first mode looks for overruns only of an entire stack frame and a heap block. The second mode looks for overruns of individual objects – it can find an overrun from one object to another within a stack frame and from one *global* to another. The first mode we call `FRAME ONLY` (despite it finding heap overruns as well); the second we will call `OBJECT OVERRUNS`. Until our recent implementation of the analyses described in this paper, the `OBJECT OVERRUN` version was not practical to run on real examples, as the false positive rate was too high. We present data for the following configurations:

`FRAME ONLY/DEFAULT BOUNDARIES` This configuration serves as a baseline for the results in this section.

`OBJECT OVERRUN/DEFAULT BOUNDARIES` The improvement to true positives relative to `FRAME ONLY/DEFAULT BOUNDARIES` illustrates the benefit to consider intra-frame overruns; the increase in false positives shows the cost.

`OBJECT OVERRUN/DDA` Relative to `OBJECT OVERRUN/DEFAULT`, the drop in false positives represents the improvement to warnings brought about by better delineation of stack objects. The drop in true positives represents cases where DDA is too aggressive in eliminating boundaries.

For all of the reported configurations, we disabled an additional warning-pruning heuristic that is enabled by default in CodeSonar/x86, called the “scalar guard.” As a result, the number of false-positive warnings reported in this paper are higher (sometimes significantly) than what a typical user of CodeSonar/x86 would currently see. The scalar guard suppresses any non-heap buffer-overrun warning where the object being overrun is four bytes or less. There are two reasons we disable the scalar guard for these experiments:

- Our long term plan is to disable the scalar guard. The scalar guard suppresses a vast majority of warnings: both false *and* true warnings. Prior to DDA, the scalar guard was the only way to enable `OBJECT OVERRUNS` without an overwhelming number of false positives, even if it did cause many true candidates to be dropped. DDA was improved and incorporated into CodeSonar/x86 specifically to support the goal of allowing us to disable the scalar guard.
- The scalar guard interacts with DDA in unexpected ways. In particular, even though the better object boundaries found through DDA “should” strictly refine the presented warnings, the number of warnings would sometimes *increase* with DDA enabled because it increases the size of an aggregate beyond four bytes. The results are not worse because we gain true positives the scalar guard suppressed, but it makes evaluating the effects of DDA. Disabling the scalar guard eliminates this interaction and leads to a much clearer picture of the effects of better data delineation.

Buffer-Overrun Detection Results. We ran CodeSonar/x86 on 68 subject binaries. Most of them – 64 binaries – are Windows programs and libraries, and the remaining 4 are Linux programs. Each Windows test is compiled twice, once in debug mode (no optimization, but still ignoring debugging information) and once in release mode (optimized). The four Linux binaries are compiled from separate programs; all four were optimized. The increase in analysis time to run DDA is negligible in all cases.

Tab. II show summary data across our benchmark set; Tab. III show full results and more information. For purposes of evaluating DDA, we are interested in the change in the numbers more than the absolute values.

We see that there is a noticeable increase in the number of matched warnings when we allow object overruns: the total number of matched warnings across the suite almost doubles from the `FRAME ONLY/DEFAULT` configuration to `OBJECT OVERRUN/DEFAULT`. However, `OBJECT OVERRUN` comes at a cost: the number of binary-only warnings climbs by about 30x.

However, turning on DDA helps substantially. The number of false positives is cut by almost a factor of 5, while the proportion of lost true positives is not much more than $1/4$. In other words, DDA allows us to retain most of the benefit of being able to find `OBJECT OVERRUNS`, while at the same time dramatically reducing the number of false positives.

Interestingly, particularly in light of the results described in Sect. IV, DDA seems to both *help more* and *hurt less* on optimized executables. Comparing the “Windows Debug” vs. “Windows Release” measures, we see that CodeSonar/x86 retains 85% of the true positives in the optimized binaries when DDA is enabled vs. only 63% of the true positives for the debug build. At the same time, DDA eliminates 86% of false positives in the release binaries vs. only 56% of them in the debug binaries. Considering that the set of programs and libraries are the same across the two sets of runs, just compiled differently, it is not clear why this happens. However, it is encouraging in terms of real-world applicability, as we believe optimized code is of the most real-world interest.

VI. RELATED WORK

Our parameter-offset analysis was inspired by the Howard tool [24, 25]. However, there are some important differences between the two tools. Howard is a dynamic analysis, and requires a test suite of (a) benign inputs that (b) collectively exercise all potential accesses of aggregate data types. Because DDA is a static analysis, it does not require a high-coverage test suite. Further, DDA infers things only from constants that appear syntactically in the code, which was driven by our position that such constants should be considered trustworthy. Howard infers some of the internal structure of objects as well as information about arrays. Our implementation finds some of this information too (e.g., it determines information about nested structures in similar circumstances as Howard), but because we do not currently need this information for our applications, we discard it.

Lin, Zhang, and Xu [26] presented a different dynamic analysis called Rewards. Rewards is primarily focused on propagating type information from type *sinks* (e.g. library or system calls) backwards and forward to locations that held or will hold a value of the same type; it does not appear to find the size of aggregate objects other than those that are passed to a known function in such a manner.

Jin et al. presented a technique for recovering information about what C++ classes and objects are present in a program binary [27]. Their tool, Objdigger, uses symbolic execution to track allocations of objects, looking at how they are passed to functions that use the “thiscall” calling convention (passing `this` in the x86 register ECX). Objdigger also recovers some information about the layout of objects based on uses within these functions. To this extent, our work resembles theirs. However, there are a number of important differences. The structure of the analysis is very different: Objdigger’s symbolic execution is done interprocedurally, while our analysis is summary-based. Our goals are very different, and this leads each analysis to infer several things that the other (in its present form) cannot. For example, Objdigger is much less flexible in the kinds of patterns it pays attention to, both in terms of only tracking objects through thiscall functions and in the fact that Objdigger isn’t prepared to learn anything from string functions.

Lee, Avgerinos, and Brumley present a technique, TIE, for performing type inference on a binary [28]. While we could probably profit from the kind of information it computes in other scenarios, TIE both does much more than we need, and yet does not quite match our targeted application domains. TIE tries to compute actual type information (e.g., signed/unsigned integers vs. pointers) whereas all we need is the locations and sizes of objects. At the same time, it appears that TIE would, at least as stated, have the soundness problem mentioned in the introduction: if a program can overrun a buffer, it seems that TIE would just infer a larger buffer size. (It would probably be possible to modify TIE to honor our “only constants are trustworthy” rule. However, if you do not need the full power of TIE, we think that our technique is simpler to understand and implement. In addition, while Lee et al. did not report running times of TIE, we suspect that DDA performs much faster.

Balakrishnan and Reps present a technique, DIVINE, for statically recovering object information from a binary [19], which is a novel combination of Value-Set Analysis (VSA) [29] and a ported version of Ramalingam et al.’s Aggregate Structure Identification (ASI) [18] from COBOL to x86 machine code. Interestingly, the goal of DIVINE in some cases is the *opposite* of our needs: a crude description is that DIVINE sometimes attempts to *separate*, e.g., fields of objects so that separate VSA abstract values can be tracked for each field (leading to more precise VSA results). ASI’s role in DIVINE is to find opportunities for summarization, but it is at a different level than is helpful for us. In addition to the different goals, DIVINE has the soundness problem discussed in the introduction – it

infers the objects based on the program’s access patterns, and thus if the program is incorrect so will the inferences be.³

VII. FUTURE DIRECTIONS

In this paper, we presented a static analysis for inferring the locations and sizes of global and local data in an arbitrary stripped executable, while avoiding the trap of choosing a “sound” technique that will infer objects that make buggy programs look correct. The inferred data layout information allows a number of existing techniques for ensuring memory safety to be applied effectively in the absence of source code. Our evaluation of the analysis showed that it achieves good precision at low cost. We used the data inferred by our analysis to significantly improve the precision of static buffer overrun detection in the defect-detection tool CodeSonar/x86.

In the future, we plan to further enhance DDA. The current work focuses primarily on detecting the layout of *top-level* objects and ignores their internal structure. We believe that our approach can be extended in a straightforward fashion to infer information about the internal structure of program data, for example structure fields that are themselves structures. (As mentioned before, we compute and discard information that naturally leads to finding internal structure.) Once we have information about the internal structure, there is the potential for finding overruns of the subobjects itself. The CodeSonar back end has support for finding warnings of this type in source code, though doing the same thing for binaries would require non-trivial extensions even once the internal structure is known.

Another improvement to the analysis that we are considering, is a more sophisticated tracking of offsets from base pointers. Currently, the analysis collects sets of constant offsets. Extending the tracking to support symbolic offsets will improve the precision of the analysis by allowing it to model naturally functions that are similar to `memset` and `memcpy` (e.g., program-specific wrappers of those functions).

Finally, most of our effort during the integration of DDA into CodeSonar/x86 was spent on reducing the number of binary-only warnings (FP proxy) with little attention paid to the matched warnings (TP proxy) that we were dropping at the same time. We would like to return to the lost matched warnings and either make sure that they are being dropped for a good reason or determine what we can do to retain them without undermining the reduction in FPs.

³VSA can be run within the CodeSonar/x86 front end, though it was not enabled for our tests. In addition, the DIVINE implementation would not *actually* create a larger object in CodeSonar/x86’s IR in the presence of an overrun – the ASI results would not be incorporated in that way.

Benchmark	#	instrs	funcs	time	DEFAULT BOUNDARIES				DDA	
					FRAME ONLY		OBJECT OVERRUN		OBJECT OVERRUN	
					TP	FP	TP	FP	TP	FP
bzip2		52,188	564	12	0	0	0	270	0	75
DXSDK.Tutorial00		29,450	439	6	0	16	0	51	0	47
freelut-2.6.0-VS9.Fractals		3,360	58	1	0	1	1	14	1	14
freelut-2.6.0-VS9.freelut		121,066	1,608	27	0	31	0	274	0	86
freelut-2.6.0-VS9.Lorenz		3,375	62	0	0	1	0	13	0	11
freelut-2.6.0-VS9.shapes		4,940	83	1	0	1	0	10	0	10
glut		56,206	815	14	15	54	15	280	8	118
libidn-1.9-VS8		9,350	137	3	2	2	2	17	2	17
libogg		33,199	504	8	0	7	0	42	0	38
libpng-1.4.0-win		58,032	418	15	3	12	6	269	0	25
miniupnpc-1.4.20100609-VS9		10,135	109	4	0	10	0	28	0	19
miranda		81,366	780	20	1	6	1	410	1	37
putty.pageant		61,130	555	16	0	8	0	269	0	115
putty.plink		121,509	1,071	37	2	12	5	359	2	170
putty.pscp		129,723	1,194	38	2	10	5	543	1	191
putty.putty		180,243	1,467	80	3	22	5	713	1	258
Python-2.1.3-win		206,617	2,395	80	12	104	31	1,297	28	358
quakeIII		341,134	3,759	159	0	66	29	2,484	10	509
repeater		30,466	373	8	0	7	2	46	2	31
sphinx.sphinx2-allphone		94,793	926	29	30	22	40	290	32	102
sphinx.sphinx2-batch		94,726	926	30	28	21	37	292	30	102
sphinx.sphinx2-client		95,763	938	31	29	21	37	296	31	96
sphinx.sphinx2-server		101,173	989	30	28	21	37	292	30	100
vpx-win-8.ivfenc		88,096	812	19	1	13	1	310	1	77
vpx-win-8.vp8-set-maps		85,680	795	17	1	13	1	320	1	74
xmail		145,910	1,597	60	42	57	62	1,371	18	107
yasm-1.0.1-VS10-x86.vsyasm		242,185	2,647	95	2	1	2	518	1	78
yasm-1.0.1-VS10-x86.yasm		242,238	2,647	104	2	1	2	523	2	85
zlib-win		20,900	141	5	1	7	10	125	10	21
Windows Dbg. C	29	2,744,953	28,809	949	204	547	331	11,726	212	2,971
bzip2		30,708	481	7	0	0	0	222	0	46
DXSDK.Tutorial00		6,979	128	1	0	0	0	0	0	0
freelut-2.6.0-VS9.Fractals		1,333	29	0	0	0	0	0	0	4
freelut-2.6.0-VS9.freelut		62,620	1,046	13	0	38	0	230	0	177
freelut-2.6.0-VS9.Lorenz		1,369	32	0	0	0	0	11	0	11
freelut-2.6.0-VS9.shapes		1,911	44	0	0	0	0	0	0	0
glut		31,033	656	7	17	7	17	177	17	60
libidn-1.9-VS8		5,431	111	1	0	11	0	19	0	18
libogg		15,847	325	3	0	0	0	3	0	0
libpng-1.4.0-win		36,484	393	8	9	0	11	216	10	35
miniupnpc-1.4.20100609-VS9		5,572	72	1	0	3	0	237	0	3
miranda		79,560	781	18	1	3	1	401	1	32
putty.pageant		37,675	453	7	2	24	2	388	2	123
putty.plink		78,980	967	20	3	35	7	654	4	244
putty.pscp		85,504	1,087	21	4	36	7	788	5	221
putty.putty		118,730	1,361	32	6	89	11	997	8	369
Python-2.1.3-win		163,905	2,339	41	16	171	29	1,576	21	588
quakeIII		233,104	3,386	91	5	90	42	4,241	39	1,036
repeater		17,260	277	4	0	1	2	51	2	39
sphinx.sphinx2-allphone		63,396	892	14	13	23	18	317	19	130
sphinx.sphinx2-batch		63,335	892	13	13	22	18	316	19	130
sphinx.sphinx2-client		64,050	903	14	13	18	19	314	19	125
sphinx.sphinx2-server		67,582	951	14	13	22	19	321	20	130
vpx-win-8.ivfenc		69,526	798	13	1	27	1	371	1	180
vpx-win-8.vp8-set-maps		66,433	779	13	1	27	1	351	1	179
xmail		111,990	1,482	35	4	24	30	2,273	8	525
yasm-1.0.1-VS10-x86.vsyasm		149,851	2,161	32	3	3	4	395	4	130
yasm-1.0.1-VS10-x86.yasm		149,867	2,156	34	3	3	4	391	4	139
zlib-win		12,232	120	3	11	1	11	49	11	11
Windows Rls. C	29	1,832,267	25,102	460	138	678	254	15,313	215	4,685
cryptopp-5.6-VS8		908,684	31,676	442	0	472	0	7,418	0	4,763
shareaza-2.2.1.0		1,117,192	29,060	662	0	136	17	3,070	7	297
tinyCAD-2.50.10		733,561	18,678	385	0	95	1	2,393	1	550
Windows Dbg. C++	3	2,759,437	79,414	1,489	0	703	18	12,881	8	5,610
cryptopp-5.6-VS8		416,109	11,774	168	0	201	0	20,943	0	996
shareaza-2.2.1.0		644,413	17,306	361	5	42	11	7,803	9	696
tinyCAD-2.50.10		386,418	15,657	182	0	215	1	4,424	1	555
Windows Rls. C++	3	1,446,940	44,737	711	5	458	12	33,170	10	2,247
Windows Dbg. C	29	2,744,953	28,809	949	204	547	331	11,726	212	2,971
Windows Debug	32	5,504,390	108,223	2,438	204	1,250	349	24,607	220	8,581
Windows Rls. C	29	1,832,267	25,102	460	138	678	254	15,313	215	4,685
Windows Release	32	3,279,207	69,839	1,171	143	1,136	266	48,483	225	6,932
All Windows	64	8,783,597	178,062	3,609	347	2,386	615	73,090	445	15,513
gnuchess-5.07		29,333	239	5	2	1	2	48	4	43
irssi-0.8.14		141,058	2,328	35	1	6	2	217	1	91
naim-0.11.8.3.1		70,044	657	17	9	32	10	154	10	61
wu-ftpd-2.6.2		27,257	201	7	3	6	13	77	9	42
Linux	4	267,692	3,425	64	15	45	27	496	24	237
All	68	9,051,289	181,487	3,673	362	2,431	642	73,586	469	15,750

TABLE III: Data for CodeSonar/x86 experiments. For summary lines, “#” gives the number of tests summarized. “Instrs” is the number of instructions, “funcs” is the number of functions, and “time” is the number of seconds spent in POA. TP and FP columns give the number of matched and binary-only warnings, respectively (our proxies for the number of true and false positives).

REFERENCES

- [1] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, “StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks,” in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, ser. SSYM’98. USENIX Association, 1998, pp. 5–5.
- [2] PaX Team, “Address space layout randomization,” <http://pax.grsecurity.net/docs/aslr.txt>.
- [3] S. Bhatkar, D. C. DuVarney, and R. Sekar, “Address obfuscation: An efficient approach to combat a board range of memory error exploits,” in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, ser. SSYM’03. USENIX Association, 2003, pp. 8–8.
- [4] S. Andersen and V. Abella, “Data execution prevention: Changes to functionality in Windows XP service pack 2,” <http://technet.microsoft.com/en-us/library/bb457155.aspx>.
- [5] T. Chiueh and F. Hsu, “RAD: A compile-time solution to buffer overflow attacks,” in *ICDCS*, 2001, pp. 409–417.
- [6] “Stack Shield: A “stack smashing” technique protection tool for linux,” <http://www.angelfire.com/sk/stackshield/>.
- [7] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Control-flow integrity principles, implementations, and applications,” *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, pp. 4:1–4:40, Nov. 2009.
- [8] U. Erlingsson, M. Abadi, M. Vrabie, M. Budiu, and G. C. Necula, “XFI: Software guards for system address spaces,” in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI ’06. USENIX Association, 2006, pp. 75–88.
- [9] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Control-flow integrity,” in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS ’05. ACM, 2005, pp. 340–353.
- [10] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, “Practical control flow integrity and randomization for binary executables,” in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 559–573.
- [11] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, “Non-control-data attacks are realistic threats,” in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, ser. SSYM’05. USENIX Association, 2005, pp. 12–12.
- [12] B. D. Rodes, A. Nguyen-Tuong, J. D. Hiser, J. C. Knight, M. Co, and J. W. Davidson, “Defense against stack-based attacks using speculative stack layout transformation,” in *Runtime Verification*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 7687, pp. 308–313.
- [13] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, “SoftBound: highly compatible and complete spatial memory safety for C,” in *ACM Sigplan Notices*, vol. 44. ACM, 2009, pp. 245–258.
- [14] —, “CETS: compiler enforced temporal safety for C,” in *ACM Sigplan Notices*, vol. 45. ACM, 2010, pp. 31–40.
- [15] N. Hasabnis, A. Misra, and R. Sekar, “Light-weight bounds checking,” in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO ’12. ACM, 2012, pp. 135–144.
- [16] M. S. Simpson and R. K. Barua, “MemSafe: ensuring the spatial and temporal memory safety of C at runtime,” *Software: Practice and Experience*, vol. 43, pp. 93–128, 2013.
- [17] “IDA Pro – interactive disassembler,” <http://www.hex-rays.com/idapro/>.
- [18] G. Ramalingam, J. Field, and F. Tip, “Aggregate structure identification and its application to program analysis,” in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’99. ACM, 1999, pp. 119–132.
- [19] G. Balakrishnan and T. W. Reps, “DIVINE: DIScovering Variables IN Executables,” in *VMCAI*, 2007, pp. 1–28.
- [20] GrammaTech, Inc., “CodeSonar static analysis tool,” <http://www.grammatech.com/products/codesonar>.
- [21] S. Tatham, “PuTTY: A free telnet/SSH client,” <http://www.chiark.greenend.org.uk/~sgtatham/putty/>.
- [22] S. Shiraishi, V. Mohan, and H. Marimuthu, “Quantitative evaluation of static analysis tools,” in *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*, Nov 2014, pp. 96–99.
- [23] G. Balakrishnan, T. W. Reps, D. Melski, and T. Teitelbaum, “WYSINWYX: What you see is not what you execute,” in *VSTTE*, 2005, pp. 202–213.
- [24] A. Slowinska, T. Stancescu, and H. Bos, “Dynamic data structure excavation,” http://www.few.vu.nl/~asia/papers/dde_tr10.pdf, Vrije Universiteit Amsterdam, Tech. Rep. IR-CS-55, February 2010.
- [25] —, “Howard: a dynamic excavator for reverse engineering data structures,” in *Proceedings of NDSS 2011*, San Diego, CA, 2011.
- [26] Z. Lin, X. Zhang, and D. Xu, “Automatic reverse engineering of data structures from binary execution,” in *NDSS*, 2010.
- [27] W. Jin, C. Cohen, J. Gennari, C. Hines, S. Chaki, A. Gurfinkel, J. Havrilla, and P. Narasimhan, “Recovering C++ objects from binaries using inter-procedural data-flow analysis,” in *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, ser. PPREW’14. ACM, 2014, pp. 1:1–1:11.
- [28] J. Lee, T. Avgerinos, and D. Brumley, “TIE: Principled reverse engineering of types in binary programs,” in *NDSS*, 2011.
- [29] G. Balakrishnan and T. Reps, “Analyzing memory accesses in x86 executables,” in *Compiler Construction*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, vol. 2985, pp. 5–23.