

Practical Considerations in Control-Flow Integrity Monitoring

Iavor Diatchki
Galois, Inc.
Email: diatchki@galois.com

Lee Pike
Galois, Inc.
Email: leepike@galois.com

Levent Erkök
Galois, Inc.
Email: levent.erkok@galois.com

Abstract—Control-flow integrity (CFI) checks ensure that programs respect their static call-graphs at runtime. A program might violate its call-graph due to malicious attacks such as shell-code injection or return-to-libc style exploits. CFI checking can also be beneficial during testing to discover properties of control-flow, as well as at deployment to detect malicious behavior. We present practical aspects of CFI checking, including advantages and disadvantages of the following: how to represent call-graphs, how to instrument CFI checks, and how to refine CFI checks to properties of control-flow. We discuss two implementations: one instrumenting the source code and the other instrumenting the compiler generated assembly, and we describe their performance. Our paper is meant to be a practical guide to CFI.

I. INTRODUCTION

Most security attacks on software aim to modify their control-flow maliciously. If the control-flow of a program can be statically obtained, then the program’s actual control-flow can be monitored at run-time to ensure conformance. This idea is known as *control-flow integrity* (CFI) [1], [2]. CFI approaches detect conventional control-flow modifications like the well-known *jumping to shellcode* [3] and *return-to-libc* [4] type of attacks, which cause the program’s control-flow to violate any possible call-graph of the program. However, carrying out a control-flow attack can be more subtle, such as modifying data values as well; CFI checks can also be used to test for these subtler attacks. Moreover, CFI checks can be used during testing to develop and check properties about the control-flow of a program: e.g., “Do the error handling routines always call the logging function”?

From a security standpoint, CFI is a draconian solution, as compared with lower-overhead approaches such as the use of canaries [5], [6], [7], address-space layout randomization [8], or custom array out-of-bound checks [9]. However, recent work shows these approaches to be ineffective at stopping all return-to-libc attacks. In a 2010 paper by Checkoway *et al.*, they demonstrate how to execute return-to-libc attacks that execute arbitrary programs without modifying return addresses [10]. The authors therefore write:

What we show in this paper is that these defenses would not be worthwhile even if implemented in hardware. Resources would instead be better spent deploying a comprehensive solution, such as CFI.

While a small number of researchers have explored specific CFI-like approaches [2], [1], there has been no discussion of some of the practical trade-offs between design choices of

CFI implementations. We discuss design choices regarding the location of CFI checks, at what compiler phase to instrument checks, the construction of static call-graphs, and how to check for conformance. Our hope is that this paper provides useful guidance for other CFI implementations for testing and security monitoring.

The remainder of the paper is organized as follows. In Section II, we overview work related work. In Section III, we describe design choices regarding various aspects of CFI, including computing the call-graphs, monitoring the control stack, and developing and testing control-flow properties. In Section IV, we discuss two implementations building on our ideas; one implementation instruments source code, and the other instruments assembly code. We also provide initial performance benchmarks. Finally, we conclude in Section V, with a brief description of future work.

II. BACKGROUND AND RELATED WORK

A large body of literature exists describing approaches for detecting, preventing, and circumventing attacks to modify the control-flow of C programs. In this section, we briefly review the work most relevant to ours. An extensive recent bibliography for CFI checking can be found in Abadi *et al.* [1]. Here, we mention some of the most related and recent work in CFI checking. We call out specific similarities and differences with prior work in the remainder of the paper.

Oh *et al.* [2] describe a CFI monitoring approach based on signature-based checks at the beginning of basic machine-code blocks. The motivation for this work is the presence of transient or permanent faults rather than security. In 2005-2009, Abadi *et al.* described an approach to CFI, where jumps in the program are instrumented with checks to ensure the targets are valid *before* jumping [11], [1]. Their call-graphs are built using static binary analysis, and instrumentation is done by machine-code rewriting, making it language-independent. Our implementations (see Section IV) shows that reasonable efficiency can be achieved with an architecture-independent (but language-dependent) approach performed during compilation. Together, the work of Abadi *et al.* and our work explore much of the design space for CFI checking.

Petroni and Hicks present an approach for monitoring control-flow attacks to detect Linux kernel rootkits [12]. Their work addresses environments in which some of the assumptions made by Abadi *et al.* do not hold [11]. The monitoring

is external rather than inline: a kernel is periodically validated from a separate monitoring virtual machine.

CFI has been applied to intrusion/anomaly detection by monitoring the call-graph to discover violations of a security policy (e.g., [13]). These efforts relate to the idea of dynamically learning call-graphs, described in Section III-C.

While we focus on run-time integrity of *control*, Loscocco *et al.* address integrity of *data* with their tool, LKIM [14]. LKIM periodically inspects the memory of its target to monitor whether static data has changed at all and whether dynamic data has changed inappropriately, possibly signaling an attack. LKIM is particularly targeted at inspecting the Linux kernel, but could be applied to other software systems as well.

III. APPROACHES TO CONTROL-FLOW INTEGRITY CHECKING

In this section, we compare and contrast different aspects of CFI checking. We begin by discussing the trade-offs of different representations of the static call-graph. Next, we address approaches to check the conformance of the control stack to the call-graph, including when and how to make those checks. Finally, we discuss how CFI checking can be used to develop more refined properties about program behavior.

The high-level property that we monitor through CFI checks is that at any point of time during the execution of a program, the program’s control stack corresponds to a valid path through its call-graph. Generally, there are two ways in which the conformance check may fail:

- The control stack might contain an unrecognized return address, which generally indicates a potential code-injection attack or “return-oriented” attack, where (portions of) known functions might be combined to cause malicious actions.
- The control stack might contain (at least) two adjacent addresses that correspond to known functions, but there is no known edge between the functions in the call-graph. This case typically indicates a potential “return-to-libc” style attack.

The basic algorithm to ensure call-graph conformance takes a control stack and a call-graph as arguments, and makes sure that the sequence of calls in the control stack corresponds to a valid path in the call-graph, starting from the `main` function.

With CFI checking, both false positives and false negatives are possible, as we discuss in the following; one important objective for CFI checking is to reduce both of these.

A. Call-graph representations

Here, we discuss three different approaches to constructing call-graphs of C-programs suitable for CFI checking. In other work on CFI, the construction and representation of the call-graph is treated mostly as an implementation detail, but its representation can affect the kinds of violations caught as well as affect performance.

As a running example, consider the code snippet and the corresponding three different call-graph representations given in Figure 1. The program snippet includes four functions that

call each other. For convenience, we substitute line numbers in the source code for return addresses.

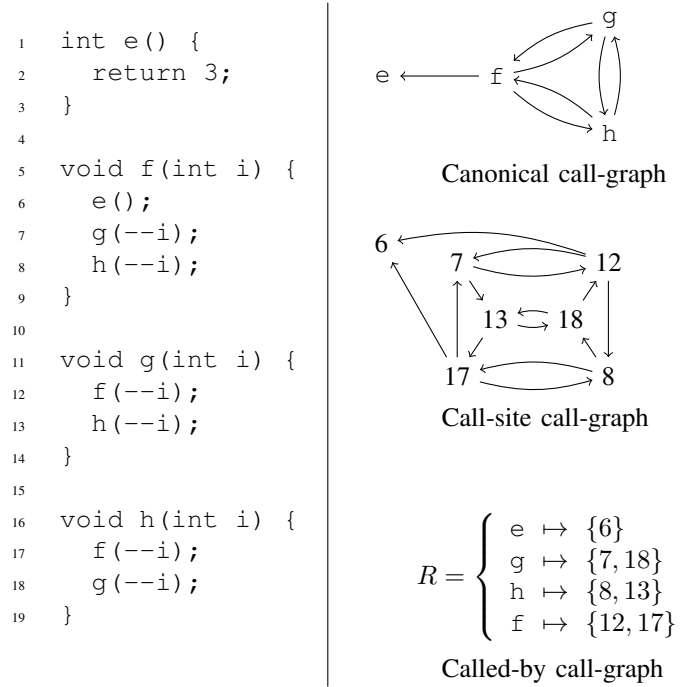


Fig. 1. Functions with multiple call-sites and their call-graph representations.

Consider the following three approaches to constructing a call-graph:

- *Canonical call-graph*: The simplest call-graph contains vertices corresponding to the *functions* in the program, and an edge from f to g if there is a function call to g in the body of f .
- *Call-site call-graph*: An alternative representation is to use *call-sites* (or equivalently, *return addresses*) as the vertices of the graph, placing an edge between two vertices if the target call-site may be *directly* reached from the source call-site, i.e., without making any intermediate function calls. (Recall that in Figure 1, we represent return addresses with line numbers.)
- *Called-by call-graph*: A third representation—a hybrid of the canonical and call-site representations—is a actually a relation R , where R maps each function f to the set of call-sites calling f . (Note that the same call-site might appear in multiple functions, if the call is made through a function pointer.)

A call-site call-graph is simple to understand and construct statically. Its disadvantage, however, is that it abstracts information about particular return-addresses within a function. For example, consider the following function, in which a message is created, encrypted, and sent:

```

1  void main() {
2      ...
3      create_msg();
4      encrypt();

```

```

5     send();
6     ...
7 }

```

Consider a CFI check made in function `send()` against a call-graph in the canonical representation: the check does not know from where in a function the call to `send()` is made, but we may wish to ensure that the call occurs *after* the call to `encrypt()`. A call-site graph provides this level of granularity. Another advantage of a call-site graph over a canonical graph for run-time monitoring is that the former corresponds directly to the return addresses available on the control stack. With the canonical representation, a translation must be made from return addresses to the function names for each monitored call, which can incur significant runtime overhead (see Section IV).

The main disadvantage of a call-site graph is that for some programs the call-graphs may be larger, as illustrated in Figure 1. As compared to the canonical representation, the number of vertices can grow at most linearly, since each function can call a constant number of functions (i.e., each function has a fixed number of call-sites). A call-site call-graph can also have fewer vertices depending on the proportion of terminal functions, i.e., functions that do not contain call-sites themselves. Indeed, in the example above, if the functions `create_msg()`, `encrypt()`, and `send()` contain no calls themselves, the call-site call-graph representation for the code snippet contains no edges at all.

The called-by call-graph representation maintains the advantage of the call-site call-graph insofar as the call-sites conform to the return addresses on the control-stack (obviating the need to translate return addresses to function names at runtime). In contrast with the call-site call-graph, the called-by representation provides a mapping for terminal functions without calls themselves. It does not distinguish callers, however, like the canonical call-graph. Custom checks can be generated statically for each function f , which eliminates the need to dynamically lookup f in the relation. Our fastest implementation uses the called-by call-graph representation.

The work of Abadi *et al.* primarily considers a call-graph R' that is the inverse of our relation R , mapping call-sites to the functions they point to [1].

Independent of which call-graph representation we choose, we will have to approximate the call-graph for some programs. This may happen if, for example, we are analyzing a program that uses a library that lacks associated call-graph information. If we do not have the source code for the library, we will not be able to construct its call-graph. A more fundamental reason for approximate call graphs is that, in general, the problem of determining what functions may be called via an indirect call (e.g., via a function pointer in C) is undecidable. As well, the use of `setjmp/longjmp` instructions can also cause call-graph inaccuracies.

For each pair of vertices in the call-graph, there are two possible outcomes of such an approximation:

- Under-approximating the edges may result in rejecting a valid control stack because an edge is missing from the

call-graph;

- Over-approximating the edges may result in failing to detect an invalid control stack because there are too many edges in the call-graph.

Whether you an under-approximation or over-approximation is appropriate depends on the context of CFI checking. An over-approximation may reduce false-positives (i.e., a violation is erroneously detected), and if the over-approximation is not gross, the probability that an attack falls within the relaxed call-graph is still low.

To improve the results of the indirect pointer analysis, we may use dynamically discovered information, while the program is executing in a trusted (i.e., sand-boxed) environment. This technique may be implemented by generating a modified version of the original program, where each indirect call-site is instrumented to record the address of the function that is being called. The information collected in this way can be used to improve the statically computed call-graph: If the set of edges is known to be an under-approximation, then we may discover additional edges. Conversely, if the set of edges is known to be an over-approximation, then we can discover edges which are known to be valid.

We return to the idea of using dynamically-learned call-graphs in Section III-C when we discuss control-flow properties.

B. Monitoring the control stack

Here we discuss when to perform CFI checks during execution as well as techniques for improving the precision of the checks.

1) *When to monitor:* We consider three possibilities for when to perform CFI checks: during function prologues, during function epilogues, and elsewhere within a function. (In the work of Abadi *et al.*, monitoring is done at the level of machine-code, providing more options for when to perform checks [1].) One benefit of prologue checks is that no matter how control-flow arrives at a function, a check is made before executing its body. The disadvantage of prologue-based checks is that attacks that “jump into” the middle of a function are not detected. To illustrate, consider the program in Figure 2. While f is executing, a buffer overflow is used to overwrite the value of a function pointer to h with an address somewhere in the middle of g . (The +39 offset in line 23 is implementation dependent.) The execution of the program thus progresses as follows: `main` calls f , where the attack is launched. Once f jumps into the middle of g , a new stack frame is *not* built for g since the function prologue is skipped. Thus, after g executes its body, it returns directly to `main` using f 's return address. In particular, the `printf` on line 25 will not be reached, but the one on line 10 (which was not intended) will be executed.

While this program's execution violates its static call-graph, a CFI check that is executed at function prologues will not detect the above attack since the attack causes a jump to the *middle* of a function. To detect such attacks, the checks should occur in function epilogues.

```

1 #define SIZE 10
2
3 void g(int a)
4 {
5     /* g is not intended to
6        be called from main,
7        but it will be. */
8     int i = 0;
9     i = a;
10    printf("Overrun!\n");
11 }
12
13 void h(void)
14 {
15     printf("Good function\n");
16 }
17
18 void f(int x) {
19     void (*func)(void);
20     func = &h;
21     void* test[SIZE];
22
23     //+39 is implementation dependent
24     test[x] = (&g)+39;
25     func();
26     printf("Unreachable!\n");
27 }
28
29 int main() {
30     f(SIZE);
31     printf("Back in main\n");
32     return 0;
33 }

```

Fig. 2. Function-pointer overwrite jumping to the middle of a function.

Two other approaches are possible in addition to—or in lieu of—checks within function prologues and epilogues. First, we may traverse the entire control stack of a program, to check that all return addresses correspond to transitions in the program’s call graph. Doing so on every function call is prohibitively expensive, but may nevertheless be useful during testing. However, a full check can be explicitly called at crucial but infrequent points in the control-flow.

determined by program design or experimentation; providing this option is particularly useful during testing. (One of our implementations described in Section IV allows for explicit checks.)

Alternatively, the monitor can be a separate program from the one being observed (the monitor must have access to the memory of the observed program in this case, i.e., it must be a privileged process). The advantage of an external monitor is that the observed program has less information about when a check of its control stack against its call-graph occurs. The greatest benefit of external monitoring is that it does not

require modification to the target. Less information makes it more difficult for a compromised program’s attacker to “clean-up” the control stack in preparation for a conformance check. The drawback of an external process is that the additional complexity of granting access to the observed program’s memory may introduce new attack vectors. Furthermore, synchronization must be maintained between monitor and the program to ensure the coherence of the observations. The external monitoring approach has been previously investigated for specialized kernel control-flow integrity monitoring [12] as well as for data-integrity monitoring [14].

CFI checks can include information about the current state of the program (e.g., the program counter, global variables, etc.) in addition to the return address. With the state, one can check not only that the control-flow conforms to the call-graph, but also that the control stack is valid for the current state. For example, consider the following function:

```

void f() {
    if (t) g();
    else h();
}

```

If the control-flow shows f calls g but $t == 0$, then the control stack is invalid.

Finally, concurrent programs do not present a fundamental difficulty, since each thread has its own control stack. In addition, in one of our implementations, we have explored the use of dedicated threads for CFI checking, improving efficiency on multicore machines (see Section IV-A1).

C. From call-graphs to properties

Even in programs that do not use function pointers, the static call-graph of a program is typically an over-approximation to its actual control-flow. The reason is that the control flow is usually data-dependent, so not all paths through the graph correspond to valid execution paths. Consider, for example, the code fragment in Figure 3.

A canonical call-graph for this program contains the path $e \rightarrow f \rightarrow g \rightarrow h2$, which is actually not a valid transition due to the particular data-dependency on this path. (Note that f is called with argument `CMD1` on line 14, and hence execution will never reach `h2`.) Similarly, the path $14 \rightarrow 10 \rightarrow 4$ in the call-site call-graph is an over-approximation.

A data-flow analysis of the program snippet in Figure 3 might suggest the following property:

“If e calls f and f calls g , then g does not call $h2$ ”

We call properties like this *trace properties*. An *execution trace* corresponds to a path through the call-graph of a particular program. We would like to specify a set of valid traces, which correspond to descriptions of valid control-stacks. One way to specify such a set is by using a pair (G, S) , where G is the program’s call-graph, and S is a set of paths through the graph that are invalid. The program monitor then needs to checks that:

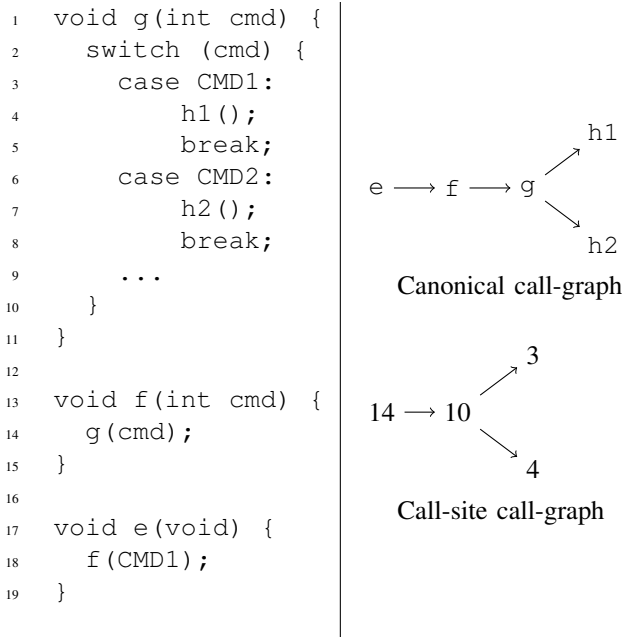


Fig. 3. Code fragment demonstrating call-graph over-approximation and two corresponding call-graph representations.

- 1) a function call corresponds to a valid edge in G (as before), and
- 2) if the edge belongs to a path p in S (the set of *invalid* paths), then check the rest of the control stack to ensure that it does *not* correspond to p .

Depending on the property, S might be compactly represented as a collection of pairs (f, g) , where f and g are vertices in the graph. Such a pair describes all paths in the graph that contain a sub-path from f to g . Thus, $(f, g) \in S$ asserts that the function f should never make a call to g , even through a transitive closure of f 's calls. In terms of an implementation, whenever we make a call to g , we should not only check that the last transition corresponds to an edge in G but, also, that f is not present anywhere on the control stack.

The effect of adding (f, g) to S may be stated precisely, using, for instance, a linear-time temporal logic (LTL) formula [15]:

$$G(f \rightarrow X(G \neg g))$$

stating, “at all program points, if f is called, then henceforth, no call to g is made”. (The formula is interpreted in a model which uses call-sites for states and control-stacks for paths. Function symbols correspond to atomic propositions, such that a proposition f holds of a state s , if s is a call-site that belongs to f .)

There are other LTL formulas that correspond to interesting properties of control stacks. For example, the formula:

$$\neg(\neg f U g)$$

asserts that the function g may be called only by f , or one of the functions that f called. Using ideas from the model

checking literature, more elaborate (and efficient) checks can be implemented using these characterizations [16].

During program execution, traces can be recorded to build a set of traces. The traces might be seeded with a statically-approximated call-graph, or with a null-graph. If the traces are recorded during testing using controlled input, then the set of traces can be used during CFI checks to see if program violates the traces observed during testing. Recorded traces can be more informative than statically-generated call-graphs. Traces correspond to sequences of valid control-stacks, making it possible to check that functions are called with the expected nesting as well as expected order. For example, Feng *et al.* describe the use of a training phase in their use of CFI for intrusion detection [13].

IV. IMPLEMENTATIONS

To demonstrate the feasibility and test the performance of our ideas, we have implemented and tested two tools:

- 1) *Source call-graph (CG) checker*: Implements CFI checks by instrumenting the source-code of a C program using a canonical call-graph, acting as a front-end to *gcc* (for creating executables) and *ar* (for creating libraries).
- 2) *Assembly call-graph (CG) checker*: Implements CFI checks by instrumenting the (x86) assembly of a program using a called-by call-graph.

After reviewing these implementations below, we present their performance benchmarks.

A. Call-graph conformance checking

To implement call-graph conformance checking, we first need to construct a call-graph for the program. This is done in multiple phases:

- 1) When we compile C source to object code, we analyze the source code to compute an (over-approximated) call graph for each file using an efficient algorithm known to work well for many programs involving function pointers [17], [18]. The result is a canonical call graph, with function names as the vertices.
- 2) When we link object files to create an executable, we also “link” the partial call-graphs into a complete call-graph, which we render as a C structure that is compiled and linked with the rest of the program;
- 3) When we package object files in a library, we also “link” their corresponding call-graphs into a partial call-graph that can be distributed with the library.

1) *Source CG checker implementation*: At run time, we support two modes of execution: manual and automatic. A manual call-graph check may be initiated by the programmer by invoking a function—linked in with the original program—which traverses the current control-stack and checks it against the pre-computed call graph. Programmers may insert calls to this function at critical program points, to ensure that the current function was called in accordance to the overall program call-graph. With just a few carefully-placed insertions, the overhead of the tool becomes nearly negligible.

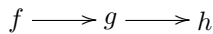
When used in the automatic mode, the prologue or the epilogue of each function is instrumented with code that will check that there is an edge in the call-graph from the call-site to the current function (which is instrumented is configurable).

We have also implemented the source CG checker to be self-checking when used in the automatic mode, by instrumenting its own source-code.

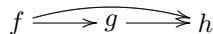
If we are using the canonical call-graph representation, then we need to map return addresses to vertices in the call-graph to perform the check. We do this with a hash-table which is gradually filled-in while the application is executing. When a return address is not present in the hash-table, we use the *binary file descriptor* (BFD) library¹ to search through the debugging information associated with the program. Once we have found the function, we look through the call-graph for a corresponding vertex. The use of BFD at run-time incurs a large performance overhead, so we cache the result in a hash-table. (Therefore, the overhead is incurred at most once per call-site. However, we should note that the one-time BFD lookup and hash-table lookups are still the significant part of the performance overhead reported in Section IV-B.) If the implementation were integrated directly with the compiler/linker, then the overhead could be avoided by constructing the hash-table statically before starting the program.

One significant advantage of this particular approach is that it is platform-independent, at least as far as *gcc* itself is. (Our implementation can be thought of as a front-end to *gcc*.) Furthermore, it easily integrates with build systems, requiring only a few small changes to typical *Makefile* based code bases.

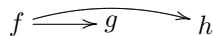
The drawback to our implementation is that some compiler optimizations may invalidate the computed call-graph. For example, consider a situation where a function *f* may call a function *g*, and *g* may call *h*:



- If the compiler *inlines* the definition of *g* at the call-site in *f*, then we need to add an additional edge from *f* to *h* because now *f* may call *h* directly,



- If the compiler decides to use a *tail-call* for the call from *g* to *h*, then we need to add an edge from *f* to *h*, and remove the edge from *g* to *h* because *h* will reuse *g*'s stack frame.



Therefore, these optimizations need to be turned off for correct construction of the call-graph when using our source CG implementation.

Ideally, the call-graph construction would use *gcc* itself. Using its plugin library should make the integration of our checks into *gcc* straightforward [19].

By default, our call-graph checking implementations, the source CG checker and the assembly CG checker that we shall describe below, check only that the most recent function call conforms to a transition in the call-graph. An alternative is to check the *entire* control-stack on each function call. Unsurprisingly, checking the entire control-stack generally incurs too much performance overhead. However, on multi-core systems, performance of full call-graph checking can be improved using parallel threads to perform the CFI checks on cores unused by the instrumented program. The idea can of course be generalized to multiple checking threads as well. In our implementation, the checking threads fully utilize their cores. A new CFI check occurs when the previous check has finished. The implementation introduces some degree of randomness for CFI checks, since when the next check is scheduled depends on the scheduling of the threads.

2) *Assembly CG checker implementation*: Our second implementation, the assembly CG checker, computes a called-by call graph by analyzing the assembly code generated by the compiler.

We use the called-by call-graph relation to generate a custom validation function for each function in the program. (We have not implemented analysis for indirect function calls in the assembly CG checker so currently, the only way to avoid false positives is by dynamically improving the call-graph as described in Section III-A.) These functions are generated just before we link the executable, at which point all the call sites in the program are known. Sample code that we generate when compiling the *bzip2* compression program [20] is given in Figure 4. Variables with CS in the name are labels for the different call-sites in the program, while the variable *ret* is the address of the caller of the function.²

```

1 void cg_bzlib_BZ2_indexIntoF
2   (void* this, void* ret) {
3
4   if (ret == &decompress_CS_6) return;
5   if (ret == &bzlib_CS_75) return;
6   ...
7
8   MSG("On entering function")
9   MSG("BZ2_indexIntoF:bzlib.s:\n");
10  MSG("Unexpected caller %p\n", ret);
11  MSG("Expected callers:\n");
12  MSG("  %p\n", &decompress_CS_6);
13  MSG("  %p\n", &bzlib_CS_75);
14  ...
15  abort();
16 }

```

Fig. 4. Sample generated code while compiling *bzip2*.

Analyzing the assembly (instead of the source) has the following benefits:

²The parameter *this* is not used. It is used to ensure compatibility with *gcc*'s `-finstrument-functions` flag.

¹<http://sourceware.org/binutils/docs/bfd/index.html>

- 1) The assembly CG checker is fully compatible with compiler optimizations because the analysis is done post-optimizations,
- 2) The tool is not restricted to programs written in C, and
- 3) Call-sites are directly referenced in the call-graph by adding labels to the assembly code.

The main disadvantage of this implementation is that, while it is not language-specific, it is platform-specific as it directly works on compiler generated assembly programs.

B. Performance benchmarks

We performed benchmark performance tests using *bzip2*, a commonly-used compression program,³ an open-source AES implementation,⁴ and a small program representing an upper-bound on the performance penalty incurred (by repeatedly making 10 billion function calls), the results of which are captured in Figure 5. The experiments were performed on a 3GHz Intel Pentium machine, running Xen/Linux compiled with `gcc`. We report the performance overhead as a multiplier of the execution times of the uninstrumented programs. The column “-O2” denotes whether the particular configuration is optimized—a configuration is either optimized with the -O2 flag or it is unoptimized. (Recall that optimizations may interfere with the Source CG checker but not the assembly checker.)

For *bzip2*, the source code is about 6.2k LOCs. There are 201 functions (including *libc* functions). The benchmarks are generated by compressing a 100MB text file. (In this case, we did not have to turn off optimizations for source CG, since wasn’t any inference.)

In our benchmarks using AES, we execute Rijndael Monte-Carlo tests at the key lengths 128, 192, and 256 (the tests are included with the distribution) 10 times each. The program itself is approximately 700 lines (not including white space), containing 63 functions, including *libc* calls. In the case of AES, the source CG checker tool has particularly high overhead when compared to the uninstrumented program since in that configuration, the program is unoptimized.

The final test executes a small program in which two functions iteratively call each other 10 billion times. Because the program does negligible computation (decrementing an integer *i* and checking whether $i < 0$) other than function calls, the program represents an upper-bound on the overhead of the implementations. The actual overhead of a typical program is typically far lower.

The assembly CG checker’s overhead is quite low—2% in the case of *bzip2* and AES. (Remember though that indirect function calls are ignored the implementation of the assembly CG checker.)

V. CONCLUSIONS AND FUTURE WORK

We have described practical approaches and two implementations for control-flow integrity monitoring. Our approach

allows us to detect a different class of malicious control-flow modifications than previous work, and may be combined with existing techniques to increase the confidence that a program is executing as intended. More generally, our work provides an efficient framework upon which to build more fine-grained run-time monitors.

There are a number of directions for future work. Developing a property specification language for monitoring control-flow properties along the lines we described in Section III-C would be a useful addition. Moreover, we have described various trade-offs throughout this paper, but we have not investigated all of them. In particular, one program monitoring another one as mentioned in Section III-B along the lines of data-integrity monitoring [14] could provide better assurance that the monitor itself has not been corrupted. Another direction would be applying these approaches to programs written in other languages.

On the implementation side, an “industrial-strength” implementation of our tools would add their functionality directly to `gcc`, so that they can be used with just an extra compiler flag.

ACKNOWLEDGMENTS

We thank George Coker, Andy White, and Grant Wagner for advice and feedback in the development of the research presented here. Aaron Tomb developed the call-graph construction library used. Xeno Kovah advised us about related research.

REFERENCES

- [1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Control-flow integrity principles, implementations, and applications,” *ACM Transactions on Information System Security*, vol. 13, no. 1, pp. 1–40, 2009.
- [2] N. Oh, P. P. Shirvani, and E. J. McCluskey, “Control-flow checking by software signatures,” *IEEE Transactions on Reliability*, vol. 51, pp. 111–122, 2002.
- [3] A. Rosiello, “The basics of shellcoding,” White paper, September 2004, available online http://www.infosecwriters.com/text_resources/pdf/basics_of_shellcoding.pdf. Retrieved January 2010.
- [4] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, “When good instructions go bad: generalizing return-oriented programming to RISC,” in *CCS ’08: Proceedings of the 15th ACM Conference on Computer and Communications Security*. ACM, 2008, pp. 27–38.
- [5] 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 *SSYM’98: Proceedings of the 7th conference on USENIX Security Symposium*. USENIX Association, 1998.
- [6] M. Frantzen and M. Shuey, “Stackghost: Hardware facilitated stack protection,” in *SSYM’01: Proceedings of the 10th conference on USENIX Security Symposium*. USENIX Association, 2001.
- [7] H. Etoh, “GCC extension for protecting applications from stack-smashing attacks (ProPolice),” 2003, available at <http://www.trl.ibm.com/projects/security/ssp/>. Retrieved January 2010.
- [8] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, “On the effectiveness of address-space randomization,” in *CCS ’04: Proceedings of the 11th ACM Conference on Computer and Communications Security*. ACM, 2004, pp. 298–307.
- [9] K. Avijit, P. Gupta, and D. Gupta, “TIED, Libsafeplus: tools for runtime buffer overflow protection,” in *SSYM’04: Proceedings of the 13th conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2004, pp. 4–4.

³<http://www.bzip.org/>

⁴<http://www.aesencrypt.com/>

Benchmark	Checker	-O2	Run-time (sec)	× Uninstrumented
bzip2	Uninstrumented	Yes	23.56	1.00
	Source CG	Yes	41.59	1.77
	Assembly CG	Yes	24.13	1.02
AES	Uninstrumented	Yes	56.08	1.00
	Source CG	No	81.25	1.45
	Assembly CG	Yes	57.13	1.02
Upper-bound	Uninstrumented	No	50.76	1.00
	Source CG	No	798.72	15.74
	Assembly CG	No	119.15	2.35

Fig. 5. Performance benchmark results

- [10] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and communications security*, ser. CCS '10. ACM, 2010, pp. 559–572.
- [11] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *CCS '05: Proceedings of the 12th ACM Conference on Computer and Communications Security*. ACM, 2005, pp. 340–353.
- [12] N. L. Petroni, Jr. and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *CCS '07: Proceedings of the 14th ACM Conference on Computer and Communications Security*. ACM, 2007, pp. 103–115.
- [13] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," in *In Proceedings of the 2003 IEEE Symposium on Security and Privacy*, 2003.
- [14] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, and C. D. McDonell, "Linux kernel integrity measurement using contextual inspection," in *STC '07: Proceedings of the 2007 ACM workshop on Scalable trusted computing*. New York, NY, USA: ACM, 2007, pp. 21–29.
- [15] M. Huth and M. Ryan, *Logic in Computer Science: reasoning about systems*. Cambridge University Press, 2004.
- [16] E. M. C. Jr., O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, 1999.
- [17] A. Milanova, A. Rountev, and B. G. Ryder, "Precise call graphs for C programs with function pointers," *Automated Software Engineering*, vol. 11, no. 1, pp. 7–26, 2004.
- [18] S. Zhang, B. G. Ryder, and W. Landi, "Program decomposition for pointer aliasing: A step toward practical analyses," in *In Symposium on the Foundations of Software Engineering*, 1996, pp. 81–92.
- [19] J. Seyster, K. Dixit, X. Huang, R. Grosu, K. Havelund, S. A. Smolka, S. D. Stoller, and E. Zadok, "Aspect-oriented instrumentation with GCC," in *Proc. of the 1st International Conference on Runtime Verification (RV 2010)*, ser. Lecture Notes in Computer Science. Springer, November 2010.
- [20] J. Seward, "bzip2 and libbzip2," available at <http://www.bzip.org/>.