# Sections are Types, Linking is Policy: Using the Loader Format for Expressing Programmer Intent

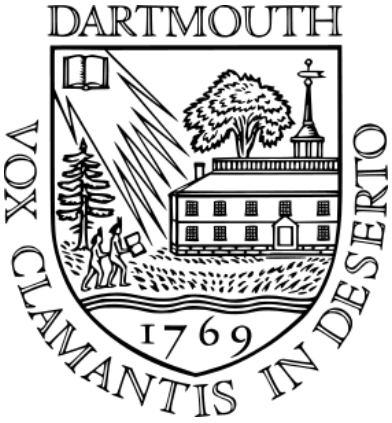
Julian Bangert, Sergey Bratus, Rebecca Shapiro, Jason Reeves, Sean W. Smith, Anna Shubina
*Dartmouth College*

Maxwell Koo
*Narf Industries*

Michael E. Locasto
*University of Calgary*

July 25, 2016

Extends and updates
Dartmouth Computer Science Technical Report TR2013-727
(June 14, 2013)



## Abstract

Attackers get software to perform unintended computation (exploits) by feeding code with data it was not intended to consume. Since security policies contain explicit specifications of intended code operation, policy enforcement mechanisms can curb exploits when exploit-induced computation runs afoul of a policy specification. However, current policy approaches leave many exposures: for example, neither filesystem-level permissions and labels nor process-level isolation protect userspace code from exposure to corrupted network inputs and from further corrupting userspace data. All these unintended interactions take place in the single address space of a process where such policies don't reach.

In this paper, we present a new approach to policy specification and enforcement *within an application's address space*. Our approach, called *ELFbac*, leverages the ELF ABI metadata already produced by the standard GNU build chain and is designed to work with existing ELF executables and to maintain full compatibility with the existing ABI. It can also be extended to other modern compiler toolchains and ABIs that support passing linking information to the OS loaders.

We show that linking data in binary components of a typical binary project contains an unmined wealth of information about developer intent. This information can be captured and made into enforceable policy with just a little help from the programmer, no changes to the C/C++ build chain or development process, and no cognitive overhead. It can also be retrofitted to old code and existing libraries—and achieve significant isolation between components not intended to interact. Moreover, the linking data structures in the ELF ABI format and similar formats turn out to be a rich channel for expressing and communicating policy to the OS loader, which can use it to set up the process address space to enforce it.

We implemented prototype *ELFbac* mechanisms for Linux on both x86 and ARM, and used them in several system-hardening projects.

This paper extends and updates Dartmouth Computer Science Technical Report TR2013-727.[1]

[1] `https://www.cs.dartmouth.edu/reports/abstracts/TR2013-727/`

# 1 Introduction

In this paper, we describe the design and implementation of *ELFbac*,[2] a policy mechanism for specifying intent of semantically distinct intra-process code units and components at the Application Binary Interface (ABI) level. ELFbac separates different code components of a program (such as libraries or functional modules of the main program) in its process' address space and monitors their accesses to their related data components, to enforce the intended pattern of their interactions—at the granularity compatible with the program's ABI format units such as ELF sections.

A typical binary Executable and Linkable Format (ELF) file on Linux contains about 30 sections. These sections are *semantically distinct* code and data units of the program; many of then have exclusive relationships, such as certain data sections being intended to be accessed or changed by certain code sections only, or to receive control only at certain circumstances and only from certain other sections. Many of these relationships come from the standard Glibc runtime, such as initialization before *main()*, relocation, or dynamic linking; others pertain more closely to the program's own code and data. Other ABI binary formats of the common COFF descent, such as Windows PE and Mac OS X's Mach-O, feature similar structures and relationships.

All of this information, however, is discarded by the OS loader—even though it clearly describes identities and intents of code and data units, and could be directly made into policy. The loader does not distinguish between ELF sections, but rather loads them in "segments", broad groups that pack sections with different semantics into a single segment. This often discards the sections' clearly expressed intent. For example, a non-executable section containing read-only data is loaded together with the executable sections—and therefore its contents also become executable at runtime, though they were clearly not meant to be so and were marked as such in the executable. Such "optimizations" are a legacy of perceived address space scarcity and smaller caches, and are largely irrelevant for modern 64-bit systems. Yet loaders persist in discarding valuable intent information and the very identity of the program's code and data units.

We present a way of "rescuing" this information for policy and acting on it for enforcement of the programmer intents. Further, we show how to augment it to express richer program-specific intents, without creating a cognitive overload for the developer, using only concepts already familiar to C/C++ programmers and supported by the GNU build chain from source code to the executable.

[2] *ELF-based access control* so named because its access control units are the native units of the Executable and Linkable Format (ELF).

We call our design *ELFbac*, as it uses objects of the *ELF* ABI as its policy principles, expressing their intended *b*ehavior in *a*ccess *c*ontrol lists (as well as by other means). We implemented ELFbac enforcement prototypes for Linux on both x86 and ARM platforms, and describe these later in the paper; yet we want to stress that we consider the design—and, in particular, its reinterpretation of the linking ABI data for policy—more important.

## 1.1 Design Goals and Contributions

Our study of the linking data and its path down the binary build chain from the source code's compilation units and libraries to the final executable in a C/C++ project revealed these as a natural vehicle for describing intents, i.e., a policy—at a minimal cost to the development tool chain and process. It convinced us that sections of the ELF binary format and comparable executable formats are *a natural unit* or *principals* for policies expressing semantic intent (we give examples of such policies below).

**Sections are natural intent-level policy principals.** Indeed, sections (a) already represent semantically distinct units of code and data; (b) can be further customized at the cost of very little programmer-supplied annotation, with no changes to the GNU C/C++ build chain, which already includes an attribute extension for creating custom sections; and (c) are intuitively clear to C/C++ programmers who understand compilation units and file-scoping. Essentially, we get an entire set of data structures that express identities and intents for free—and also communicate them to the kernel (where the enforcement mechanism resides) in its *native format* for describing and handling binary objects.

**[Most] sections are types.** Generally speaking, the idea of types as data abstractions is that of a collection of objects such that a particular set of operations can be applied to each object [14]. It is, of course, a stretch to extend it to all ABI sections, but many are indeed defined by their exclusive relationships with some specific code; thus we have their intent and identity defined through operations on them, at least with regard to access. Indeed, whenever a data section can be defined in terms of exclusive access from some code operating on it, it should be, as it may help prevent unintended computation (which we discuss in the next section). This relationship can be seen as biased towards data sections, but it is, in fact, symmetric: data flowing from unintended units to code is a security risk, because that data was likely not validated for that code and may cause unintended behaviors (the UDEREF feature of the Grsecurity/PaX Linux hardening suite protects the kernel against just such poisoning by user data).

**GNU build chain already supports lightweight annotation.** Our annotation typically requires no more than an extra line per compilation unit; it is supported by GCC "out of the box". Custom sections can be created with the GNU pragma (`__section__(...)`), and are handled transparently by the linker. If needed, these can be even be positioned at specific memory locations by manipulating the linker map, a trick familiar to (at least) embedded programmers and intuitive to others.

**The loader is a key policy enabler.** The only part of the platform where substantial changes are required is the OS loader. Their main purpose is to maintain knowledge of where the executable's and the shared objects' sections are mapped within a process space, and to arm the virtual memory system to trap accesses that are contrary to the sections' respective intents—turning the loader from a "forgetful" one into an "unforgetful" policy helper. Since the loader is the OS component dedicated to parsing and handling an executable's structure, it naturally becomes the centerpiece of the enforcement mechanism for the executable's parts. It already does much of this work by design—and it's only natural to extend it to do more.

**Linking is policy.** Grouping of code and data with similar semantics together in a binary enables common access protections for the group. Thus grouping—performed by the linker, and not paid much attention to—is actually not just a convenience but also a policy enabler, and should be treated as such. The linker then becomes an expressive policy tool.

**Sections want to be connected with VMAs.** From the data structure-centric point of view, both ELF metadata and the kernel's virtual memory area descriptors (e.g., `vm_area_struct` structures) contain very similar information describing the identities and properties of dedicated contiguous areas within a process' virtual address space. It's just during loading that these identities are handled a lot more coarsely, for reasons no longer relevant to modern systems. ELFbac's unforgetful loader bridges the current unfortunate gap between these two kinds of closely related data structures.

**How to read this paper.** Section 2 discusses how the policy phenomenon of programmer intent relates to exploitation and its mitigations. Section 3 makes the case for intent-based policies to address composition. Section 4 gives a motivating example, and Section 5 explains our approach to *intent-level* policy.

Section 6 presents our design, and Section 7 discusses how applications and OSes could best take advantage of it. Section 8 discusses our prototype implementations. Finally, Section 9 sketches out ways to evaluate its real-world effectiveness.

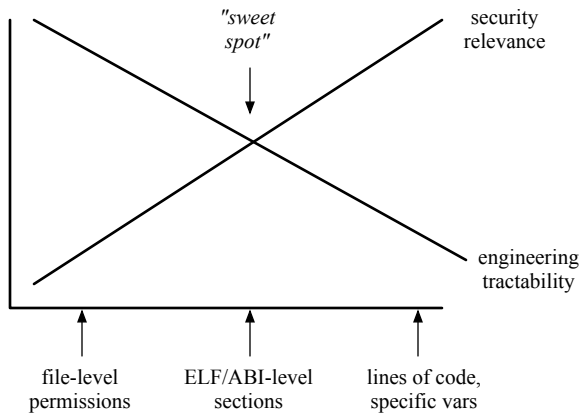



Figure 1: The ABI-level code/data granularity is the "sweet spot".

## 2 Intent vs Unintended Computation

The point of policy is to prevent unintended computation.[3] By contrast, exploitation is inducing unintended computation on the target; it has been described as discovering, instantiating, and programming an unexpected computation model (a so-called "weird machine" [18, 40]) inside the target.[4] The specific "machines" underlying unintended or unexpected computations reused code from traditional C/C++ compiled implementations of the control flow between functions via a call frame stack (ROP, JOP, [38] etc.), heap management for DL-malloc-derived and Jemalloc heaps [3, 4, 26], Unix signals (SROP [10]), implementation of virtual functions in OOP (COOP [39]), RTLD metadata [34], GNU C++ exception handling [32], x86 MMU descriptor tables [8], and so on.

Unintended computations widely differ in their tasks, complexity, and in the kind of target code they reuse. For example, a browser heap exploit may chain several information leaks and memory corruption bugs, whereas Heartbleed relied on a single parser bug not involving any corruption. But whether the vehicles of unintended computation and the programming of the weird machine depend on memory corruption, manipulation of metadata, or corruption-less misinterpretation of inputs[5]—

[3] There is a notational difference between *unintended* vs *unexpected* computation: some kinds of unintended computation are expected though must still be prevented. Others are neither intended not expected; they are the "unknown unknowns" of the policy. We'll use unintended to mean either.

[4] The exploit itself serves as a proof-by-construction of the unintended computation model's existence and programmability by attackers.

[5] Prominent examples of these are SQLi, XSS, parser differentials such as X.509 [24] or Android Master Key [22], and, lately, Heartbleed and Shellshock, among others.

*their common feature is that the programmer's intent is violated.*

## 2.1 Infer or Specify?

Consequently, security policies seek to constrain computation to its intended constraints. The key question is whether to *infer* intent or to let the programmer *specify* it explicitly—or use some combination of both, to avoid overburdening the programmer.

**Infer.** Actual binary execution in common platform could stray very far from the language-based abstractions and related invariants that a programmer would take for granted. Common C/C++[6] compilation conventions and userland runtime mechanisms such as heap management and even input-output systems [30, 36] used to place no restrictions at all on either memory accesses or on control transfers within x86 userspace.[7] For example, returns or jumps would land on instructions never meant to be a target of one, or, in fact, inside a multi-byte instruction, causing instructions never emitted as such by the compiler to be decoded and executed; heap management code would splice "free lists" of chunks in memory areas other than the heap; stack operations would occur on "pivoted" memory areas other than the stack; a dynamic linker-loader would "relocate" bytes in memory segments never meant to be relocated [34,41], and so on. Even after the introduction of hardware primitives such as the NX bit and SMEP/SMAP for trapping processor accesses to certain memory areas contrary to the systems programmer's intentions for them,[8] the gap between programmer abstractions and the actual runtime execution environment remained huge (and encouraged mitigations such as code first checking for signs of known integrity violations in the data before operating on it).

Thus a variety of approaches arose to infer the natural constraints derived from these abstractions—assumed to be unambiguously included among the programmer's intents—translate them to systems primitives such as trapping or memory translation, and enforce them as a part of the policy. Among these are Control Flow Integrity, Data Flow Integrity, Software Fault Isolation and its variants (since software modularization is another abstraction programmers are expected to understand and intend), XFI, and others [1,16,20,21,25,27,42,45]. Powerful techniques like memory shadowing and dynamic taint propagation were developed to support their implementation. Considering the gap between the possible in common platforms—especially x86—and the higher-level abstractions in which the next generations of programmers are taught, these translational approaches will continue to be productive.

However, these approaches have one thing in common: they leave the programmer—and thus the programmer's knowledge of what is intended beyond the programming language abstractions—out of the picture.

In a perfect world, the language's own constructs would go a long way. For example, translating the functional languages maxim, *Make illegal state unrepresentable*, down to hardware in a strict fashion would kill most exploit techniques because their execution literally depends on a sequence of illegal states [18]. However, functional languages for which such translation would be most effective are the furthest from production.

This cuts both ways. Whereas the programmers aren't burdened with formulating any additional expectations, they may also know something about the semantics of the data that goes beyond the language abstractions. For example, the programmer might know that the cryptographic keys are the crown jewels of a program and should only be operated upon by certain code. Language abstractions, even in a language with concepts of encapsulation and isolation, may not be flexible enough to express and prioritize this knowledge for an automatically inferred policy.

In our design, we infer over a dozen of access rules and relationships between the standard units of the Glibc runtime, including the relationship between the dynamic linker (`ld.so`, the Linux RTLD) and the Global Offset Table (GOT) sections of any loaded executable and shared object (the GOT is the only section of the executable space that the RTLD is intended to write, although it can be tricked into overwriting many other locations [7]). Similarly, we infer these relationships for any loaded shared object where they exist.

It should be noted that most of the unintended computation mechanisms mentioned above—notably, except those that depend on the parser misinterpretation of crafted inputs—involve referencing of data units in memory by code units not meant to do so. Whereas all such relationships would be hard to describe, at least the most critical ones for the security of a program can be expressed and prevented by grouping code and data into named sections with exclusive relationships.

**Specify.** Letting the programmers enforceably specify their additional knowledge about the semantics of a program or its units, is, of course, a major area where programming languages meet systems research, and is too

[6] "C is a programming language for turning byte arrays into security advisories", Francois-Rene Rideau, `https://twitter.com/fare/status/657349102078984193`

[7] Other platforms had some weak restrictions like address alignment—which still made ROP for RISC and ARM non-trivially harder than for x86, cf. [19].

[8] We note that enforcing these intentions was a part of advanced defense strategy even before these primitives became available; the Grsecurity/PaX pioneered their semantics in NOEXEC and UDEREF features by cleverly emulating these primitives long before they became available—and essentially made the industry case for their usefulness.

broad to review here. In practice, however, it poses hard challenges for use in production.

What are the units in which the semantics and the intents are specified? How do they translate down to the system's ABI elements? Most importantly, can the programmers be persuaded to annotate their programs, let alone learn a new language? Can the annotated code coexist with non-annotated one in production?

A policy that requires too much additional specification is likely to see slow adoption. For example, SELinux policies required the programmer or the administrator to specify all allowed accesses via file and process labels—a burden in the traditional Unix model. As a result, few Linux distributions adopted strict SELinux policies. It took the introduction of Android with much more restrictive defaults of programming model for SELinux to become a part of a widespread Linux platform.

Not surprisingly, the semantic annotations that did get broadly deployed came in as seamless extensions of the existing build chains rather than as new languages or subsystems. For example, the now-common annotation of userland pointers in the Linux kernel depends on GCC's attribute extension.

In our design, we build on the existing GCC attribute extension, by use of which the programmers annotate the program's code and data units, and relationships when such annotations are warranted by the special role of the units or their exclusive relationships.

## 3 Composition

Modern programs are overwhelmingly built by composing libraries, modules, classes, and objects into some cohesive whole. Each component, composed of code and/or data, has some explicit high-level purpose or *intent*. The intended use of a component defines its *expected behavior*, not least w.r.t. its accesses to other components. It's hard to estimate where, in what part of actual exploitation *incidents*, the unexpected computation caused by the exploits violates programmers' intended cross-component interactions, but many exploitation *methods* indeed do so.

For example, the intended usage of the `libpng` library is to convert compressed images to bitmaps; the intent of a table of virtual function pointers is to be set up at the beginning of a process' execution and called throughout, the intent of the Global Offset Table (GOT) is to be written only by the dynamic linker-loader (RTLD). Note that the function pointers or GOT entries should *not* be overwritten by `libpng`—but that's what an attack exploiting a vulnerability in `libpng` may do. Similarly, a library tasked with parsing data from the Internet (such as a DNS resolver, `libresolv`) is not intended to read user private keys kept by an SSL/TLS library—but may in fact do so. [9]

Modern ABI formats descended from COFF, such as ELF, PE, and Mach-O[10] already distinguish up to 30 semantically distinct types of code and data sections in a typical executable, and allow programmers to easily define more custom types.[11] Most of these types can be characterized by their intended mutual relationships ("code section X is only meant to read data section Y and write Z"). However, at runtime all such information, although available to the loader, is forgotten—once loaded, all code can access all data within the process, and pass control to any other code within the process, no matter what the original intent. ELFbac changes this loader behavior.

ELFbac's loader (we call it an *unforgetful* loader and discuss it in Section 7.5) remembers the identities of the ELF sections it loads,[12] and, in our x86_64 prototype, enforces the access control relationships between them using the x86 MMU's virtual memory support. Accesses contrary to the policy are trapped and handled; the list of intended (allowed) accesses is stored in a special ELF `.elfbac` section.

ELFbac brings policy to the exact same basic level on which modern software is composed—and allows programmers to express the intent of the components with a process. We observe that data structures in modern ABI formats provide a natural, flexible, and effective way to express and label semantically distinct code and data components that act as subjects and objects of intra-process, inter-component access control policies. Thus our prototype expresses *intent-level semantics at the ABI granularity*.[13] As Figure 1 illustrates, this is the "sweet spot" for policy enforcement as it allows for detailed, yet not burdening communication of a component's intent from its programmer to the enforcement mechanism

[9] A somewhat compressed list of `libpng` vulnerabilities: CVE-2006-{0481,3334,5793}, CVE-2007-{2445,5266,5267,5268,5269}, CVE-2008-{1382,3964}, CVE-2009-0040, CVE-2010-1205, CVE-2011-{0408,2690,2691,2692,3026,3048,3328,3464}, CVE-2012-3386, CVE-2013-6954, CVE-2014-{0333,9495}

[10] Although in this paper we focus on ELF as the dominant format of the Unix OS family, our ideas can be applied to other types of executable formats and operating systems.

[11] The GCC toolchain already supports creation of custom sections from C/C++ with the GCC extensions pragma `__section__`.

[12] Unix loaders traditionally operate on *segments* and disregard *sections*. This helped conserve address space fragmentation, arguably important for 32-bit address spaces (at the cost of mapping, say, `.rodata` as executable; these days, 64-bit address spaces obviate the need for such economy.

[13] In standard theory usage, program semantics are described in terms of predicates, invariants, and various formal logic statements. Such formal tools, however, are typically outside the reach of an industry programmer, at either developer or architect levels. We do not intend to use the term in this meaning and instead focus on high-level properties and statements that match common programmer intuitions.

throughout the binary toolchain [12].

## 4 Motivating Example

Large numbers of software vulnerabilities are exploitable because code and data that live within the same process address space can interact with each other in ways not intended by the developer. We consider a simple example of unintended interactions to highlight the shortcomings of current approaches to access control.

Figure 2 shows some C source code for a file server that reads in requests for files, fetches the files from disk, encrypts the files, and sends the encrypted files over the network.

Unfortunately, `process()` has a buffer overflow bug. Due to this bug, an attacker with control over the input to the file server can write to arbitrary memory locations. The attacker can use this ability to change the flow of control in the server. If the server uses defense mechanisms such as the NX-bit preventing code injection, the attacker can use techniques such as return-oriented-programming [29, 38]; if the server uses the defense address space layout randomization (ASLR), the attacker can use techniques such as Müller's [28], etc.

In a nutshell, not knowing a memory address is not the same as being unable to access its contents, because a symbol not "officially" exported may still be computed, as it happens often enough via information leak bugs. We note that verifying the absence of info leaks in software is a daunting task; a defender able to trap unintended accesses appears to have a much stronger position that one who depends on the attacker being unable to compute what to access.

Once the attacker alters the control flow via this bug in `process()`, the attacker can proceed to the "crown jewels" in `encrypt()`, such as disabling encryption or extracting the encryption key.

It is not unreasonable to assume that `encrypt()` was provided by a well-vetted third-party library. Finding a cryptographic vulnerability in such libraries is non-trivial; however, simple bugs that leak the cryptographic key material are not exactly rare (cf. Heartbleed, the GNU TLS Hello bug, and others). Since all code and data elements in a process live within the same address space, the key material can be leaked through exploiting the `process()` function, which probably would not have been (and should not have needed to be) analyzed by the more security-aware engineers who implemented `encrypt()`.

```
static char *encryption_key = "super secret";

void input(){
  int *data = read_data();
  process(data);
}

void process(int *data){
  data[data[0]]= data[1];
  /* We have a trivially exploitable
   * write-1-byte primitive here.
   * In the real world, this would be hidden
   * or constructed through smaller primitives
   */

  void *encrypted_data = encrypt(data);
  send(encrypted_data);
}

void *encrypt(int *data){ ... }
void send(void *data){
  printf("%s", data);
}
```

Figure 2: Simple program demonstrating the problem of too-coarse protection granularity: the vulnerability in `process()` compromises the secret key used by `encrypt()`, even if `encrypt()` is a separate and well-vetted library.

## 5 Our Approach

Deployed Unix policies, up to and including SELinux and AppArmor, treat an application as having a "bag of permissions", which can be exercised by its process in any order and any number of times. However, there is a clear benefit for, within an application, not treating *every segment of code* (including all the libraries) as equal. Just because fork(), exec(), and `memcpy()` are mapped into a process's address space doesn't mean, e.g., that security-critical code that validates untrusted input should be able to invoke these functions at any time.

The labeling of code and data units for programmer intent and expectations must exist and be enforced on the same level where the primary act of software engineering—*composition*—takes place. Security labels must match the units in which software is written, imported from libraries, and, ultimately, loaded and linked at runtime. ELFbac flexibly supports labeling at different granularities, from individual symbols (treated as a section) to whole libraries, so a policy can be expressed at the granularity that makes the most sense to the developer.

These days, it is practically impossible to write a meaningful program without using code from scores of different external libraries, likely written by third-party

developers. Even if the original libraries were vetted in some way, they will likely be updated or replaced as the underlying operating system and system configuration changes. Furthermore, even first-party code is likely also structured into different modules, each with its own intended use. For example, consider the modern Firefox browser. While it is running in Linux, over 100 libraries are mapped into its address space and can be accessed from any code executing in the context of the Firefox process. Examples of libraries mapped into the Firefox address space include `libFLAC.so` (encoding/decoding certain types of audio files), `libdrm.so` (direct access to video hardware), `libresolv.so` (handles DNS lookups), `libssl3.so` (SSL and other crypto), and `libc` (standard C library functions). Each library has its own purpose and is used by Firefox with a certain intent in mind. However, a bug in one library (for example `libdrm.so`) can be triggered by a second library (for example, `libresolv.so`)—even if, from the developer's point of view, the second library has no business accessing the first library. Since Firefox uses each of these libraries with a certain intent in mind, it is natural to draw trust boundaries around each library and within the main executable itself, treating each segment of code and data within a boundary differently from a security perspective.

There are also other potential natural boundaries at various granularities we can consider—including functions and object files, as each is created with a certain intent in mind. However, all these things—functions, object files, libraries—have two key things in common. *(I)* each reflects some separate intended semantics ("what this code is for"), and *(II)* the compiler-linker toolchain has knowledge of each during the build process.

In ELFbac, we use (II) to capture (I), and extend the chain to the OS loader, making it "unforgetful" of this rough semantics of intent. Examples include "this library function is only meant to work on this kind of data, type of object, or region of memory", "this library function is only intended to be executed during initialization phase of a program or during error-handling", or "these data are only meant to be accessed in the initialization phase of a program or during error-handling".

# 6 ELFbac Architectural Design

Figure 3 shows the overall architecture of our ELFbac prototype. Section 6.1 presents our policy language. Section 6.2 discusses an ELFbac policy for our example from Section 4. Section 6.3 presents tools to assist a programmer in creating a policy.

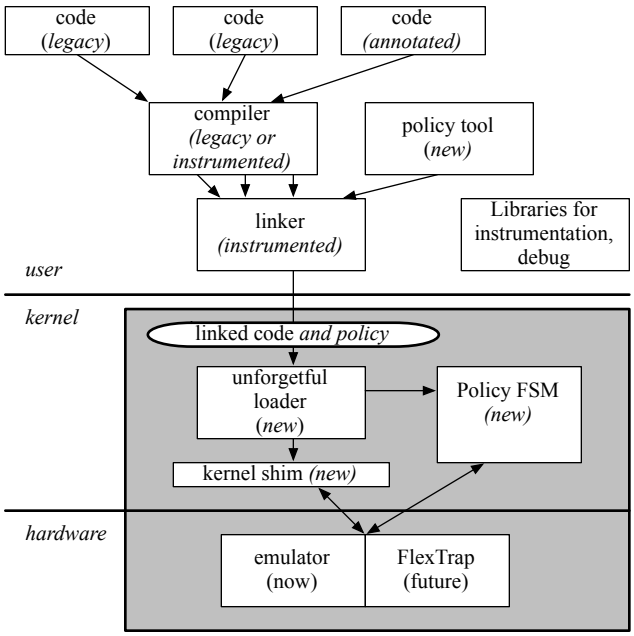



Figure 3: The ELFbac architecture

## 6.1 ELFbac Policy

ELFbac policies are expressed as finite state machines (FSMs). Each state is a label representing a particular abstract phase of program execution that a given section of code in the program or library drives. The inputs to the automaton are memory accesses of the program. The policy defines a set of transitions between these states (such that there is at most one transition for each pair of state and memory address) and the allowed unit-level memory accesses for each state.

Practically speaking, being in a particular state means that (1) the Program Counter (`RIP`) is currently inside a particular section, and (2) the program's control flow transitioned from an appropriate previous state (or the initial state). Thus a code section can be visited in more than one state, depending on the program's execution history. We note that the program counter position as an indicator of the intended program state has been used in several projects, such as Sancus [31] and Salus [6], where it is called *PCBAC*[14]; however, associating multiple states with the same PC has not, to the best of our knowledge, been explored.

In particular, for each state in an ELFbac policy, there is a set of rules that specify what data and code can be used, and how these sections can be used in terms of read, write, and execute permissions. In this manner we are able to treat different sections of executing code within a single process differently with respect to secu-

[14] `https://distrinet.cs.kuleuven.be/software/pcbac/`

rity. These rules also need to specify what state to transition to after a given memory access, which often is the same state the process was in prior to the memory access. In a graph representation of the FSM, the transitions that do not change state correspond to loops, whereas those that do correspond to non-loop edges, so all information is encoded in the edges of the graph. In fact, the statements of our policy language are isomorphic to directed edges of the FSM.

```
policy_statement :=
      data_rule | call_rule
```

An ELFbac policy distinguishes between two kinds of rules: *memory access* rules and *function call* rules. Each policy rule is a transition in the finite state machine, so we call them "data transition" and "call transition" respectively. Each transition specifies a source and destination state and the interval of virtual addresses that trigger it. In any given state, there is exactly one rule that specifies how to handle a memory access to a given offset. If no rule is explicitly specified, the memory access is denied, which usually results in a segmentation violation and the (logged) termination of the program. If a rule does not specify a destination state, it is assumed that the destination state equals the source state.

```
data_rule :=
   from_state (-> to_state)
      {read,write,exec}
      from_symbol (to end_symbol)}
```

Data transitions allow the program to read, write, or execute the specified memory address. We expect most rules regarding data accesses to not cause state transitions—for example, "Code executed while in state *cryptography* is allowed to read memory labeled *key material*". However, rules that transition to a different state on a memory access attempt are also valid. If only a single symbol or address is specified, this corresponds to the implicit 'size' of the referenced object, as determined by the ELF symbol tables and headers. ("Code executing while state is *input* may trigger a transition to state *cryptography* when reading memory labeled *key material*.")

```
call_rule :=
   source_state -> dest_state
     call symbol (return) (paraminfo)
```

A call transition does not allow memory accesses, yet its semantics make expressing transitions between states more convenient. It triggers on an instruction fetch from a single virtual address, usually corresponding to the entry point of a function. In addition to syntactic convenience features for parameter passing, call rules can also allow a one-time return transition back to the source state. Return transitions allow access to the instruction after the jump that went to the faulting instruction, corresponding to a function return.

If function returns were not treated differently, then a widely-used function (for example, in the C library) would have to be allowed to jump to the instruction following every call to it, which would give an attacker threatening flexibility in changing the control flow.

However, both the call and return transitions are merely convenience features and can be replaced by multiple states with data transitions when reasoning about the policy. For example, suppose we have states `X,Y,Z` with:

```
X exec x_func
Y -> X call x_func
Z -> X call x_func
```

We can then equivalently create two states `X_1,X_2` with:

```
X_1  exec  x_func
X_2  exec  x_func
Y -> X_1 exec x_func-x_func
Y -> X_2 exec x_func-x_func
```

If this call rule had return transitions, those would correspond to

```
X_1 -> Y exec y_ret
X_2 -> Z exec z_ret
```

where `y_ret` and `z_ret` are the instructions following calls to `x_func`.

By removing these convenience features, the ELFbac policy can be reduced to a regular expression matching the memory accesses of a program. Whereas it is very hard at best to make any useful statements about the machine code itself (as it is Turing-complete), we can make statements about the policy, as it's a weaker finite state machine. For example, it would be very convenient to be able to prove a statement such as "data from the filesystem must be encrypted before being sent over the network". In general, proving such a statement about a program itself is undecidable by the Rice-Shapiro theorem [35]. However, assuming we can trust our cryptography and authentication code, we can put all code writing to the network in one state and then verify that all transitions leading to that state occur from the cryptography state. ELFbac then guarantees that all data will have been passed through the cryptography module before it is sent, reducing the amount of code that has to be formally verified.

## 6.2 Policy Example

Our example program in Figure 2 can be naturally divided into four phases: input, processing, encryption, and output. In a reasonably well-designed program, these four phases would be in different modules, functions, or classes, depending on the language in which

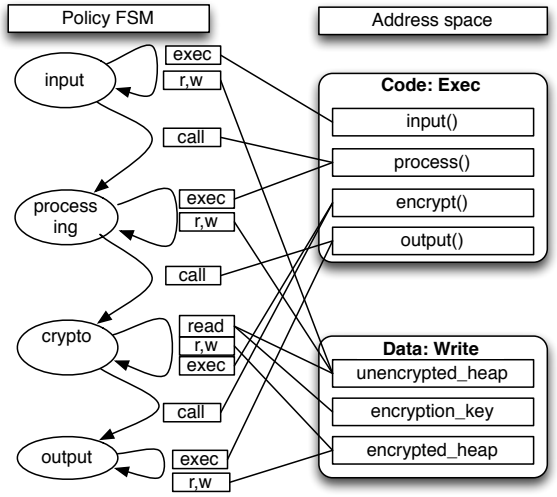



Figure 4: A sample ELFbac policy for our program example from Figure 2.

```
HEAP unencrypted_heap, encrypted_heap
input_phase exec input
input_phase read,write unencrypted_heap
input_phase syscalls * //Allow all syscalls
input_phase -> processing_phase call process

processing_phase read,write unencrypted_heap
processing_phase exec process
processing_phase -> crypto_phase call encrypt

crypto_phase exec encrypt
crypto_phase read .encryption_key
crypto_phase read unencrypted_heap
crypto_phase read,write encrypted_heap
crypto_phase -> output_phase call output

output_phase exec output
output_phase syscalls *
output_phase read encrypted_heap
```

Figure 5: Source code of the sample ELFbac policy.

the program was implemented. (We posit that, for programs in general, security-relevant progressing steps can be similarly separated into such stages.)

Figure 5 is such an ELFbac policy as the programmer creates it. Figure 4 shows this policy as a finite state machine. Each arrow label corresponds to one transition, which is triggered by access to a particular region in memory. Without ELFbac, only writable data and executable code are separated by the memory system, as shown by the shaded boxes.

The four states are only allowed to execute one of the code modules at once. Similarly, each has different access permissions with regard to the three data areas. The input stage is allowed to access system calls[15] and write to the unencrypted heap area. The processing phase, which contains the critical security vulnerability, can only read or write to the unencrypted heap and neither affect the key material nor cause sensitive data to be sent unencrypted.

## 6.3 Automating Policy Creation

We automatically generate policies to isolate an executable and the libraries with which it is dynamically linked, by default creating two states, one for the libraries and one for the main binary.[16] Our tool analyzes ELF symbol tables to determine which imported library functions get called and creates a call rule for each of those. So far, the programmer needs to manually allow callbacks and other instances where the library should be allowed to call a function in the main program.

Because it is difficult to determine control flow within ELF sections or which data should be accessible by which code from binary code alone, more advanced tools are under development.

# 7 Designing for Intent-level Semantics

Our goal is to help humans and systems better reason and communicate about trustworthy behavior. A basic characterization of a trustworthy system is "the system and all of its components behave as *intended* and *expected*." Thus, to build a trustworthy system, it is crucial to be able to express and communicate both *intentions* and *expectations* as well as verify their congruence (or at least compatibility). One of the simplest and yet most important elements of this communication is specifying *intent* of a particular software component—that is, a high-level description of *what it is meant for*.

An application's developers typically have the best understanding of what the intended behaviors of the application's code units are, even within large multi-developer applications. An application's developers are also best suited for picking out what variables/symbols/data/control flows are most sensitive and important with regards to security. For example, the developers tend to have the best idea of which sections of code should be off-limits until a user has authenticated, and which sections

[15] System call restrictions were not used in our results section, as we are currently developing mechanisms that are portable between architectures and do not impact the existing kernel design.

[16] Every library should have its own state, and if the user specifies this, such a policy can be created. However, some libraries (such as *pthreads*, *dl*, and *libc*) interact in ways not obvious from their ELF symbols.

of data should be off-limits to external helper libraries[17] throughout the program's runtime. Therefore, we have designed ELFbac with the purpose of allowing an application's developers to express an application's intended behaviors in an ELFbac policy in a concise yet usable manner.

## 7.1 Co-designing Application Structure and Policy

We believe the solution is to move toward compartmentalizing segments of code and data inside a process's address space. Our approach is to use the code and data granularity *already present* in the loader format to specify intra-process security policy, and then to augment the existing software and hardware architectures to enforce it. We believe that this approach will lend itself to developing policies supporting intent-level semantics.

We need to be able to create policies that support intent-level semantics on the appropriate code and data granularity. To achieve this goal, we rely on a series of insights:

- Code and data are already semantically divided at development.
- These code/data divisions are conveniently carried over to the binary level by the loader format specification.
- Even if the developer does not explicitly specify a policy, we can often infer an implicit policy from the code/data divisions already in the executable's ELF section headers and symbol tables.

Developers tend to structure their applications into different modules and formulate "interface contracts" between the modules that specify the intent and behavior of each unit of code. Such rules/contracts can be explicitly encoded in object-oriented languages such as C++ and Java, where classes, fields, and methods are marked as public, private, or protected. It is up to the compiler to enforce these rules—unfortunately, *this knowledge is discarded and thus not enforced at runtime.*

Since software is designed in terms of libraries, modules, and individual symbols, our policy mechanism targets these as granularity levels. We focus on the binary toolchain because it operates at precisely these granularities. Libraries and modules correspond to different ELF files, and most compilers allow the programmer to arbitrarily group symbols into sections through vendor-specific extensions at granularities that make the most sense to the developer. For example, in Figure 2, the programmer can write a very fine-grained policy for security critical components such as the `encrypt()` function, yet treat the entire `process()` module (which in a real world application would consist of many functions) as a single entity.

[17] Attackers leveraged vulnerabilities in popular image and media libraries to get through them at the application's "crown jewels" data, even though those libraries were obviously not intended to access such data, only to render generic presentation material.

## 7.2 A Note on Semantics of Unit Relationships

It's also important to consider the *unit relationships* between code and data. In a well-designed program, exclusive access relationships alone can serve as a good proxy for the unit's intent—and therefore as a basis for succinct expression of an enforceable access policy. (As we show later in Section 7.4, the standard ELF section structure already expresses a wealth of such relationships.)

Due to the inherent complexity of modern applications, we treat code units as black boxes whose intended behavior can only be verified by their interaction with other code and data units. Non-trivial semantic relationships can be extracted from mere annotations of intended accesses, just as a grammatically correct sentence made of nouns of obscure or undefined meaning nevertheless conveys important information about the referents of these nouns. A classic example is *"the gostak distims the doshes"*[18]: even though we don't know what the *gostak* or the *doshes* are, we know of their relationships that the *gostak* is that which *distims* the *doshes*, that *distimming* is what the *gostak* does to the *doshes*, and so on. For all intents and purposes, we can now define a policy to describe this as an intended behavior, so long as we have labels for *gostak* and *doshes*, and can recognize *distimming.*

A complex program contains many code units that will likely remain as opaque to a policy mechanism as the *gostak* is to English speakers, and whose intended behavior can only be described based on their relationships with other units. This opacity strongly correlates with the amount of context needed for a code unit to determine if the data it is about to work on is valid. For a parser (or an input recognizer in general), it is comparatively easy enough—or should be easy enough with the right messages format design—to check if a particular input bits or bytes are as the input language grammar specifies, based on locally available context (e.g., a few bytes previously seen in input or a few bytes of look-ahead).

Unfortunately, not all code units are so comparatively lucky. Consider a typical collection data structure such as a heap or a hash table of C structs or C++ objects that are chained (via pointers or offsets) internally and

[18] The phrase was coined in 1903 by Andrew Ingraham, and popularized by C.K. Ogden and I.A. Richards in their book *The Meaning of Meaning.*

with other objects; and consider code units that follow the pointers and change the collection elements and their linkages. For such a structure collection to be valid ("as expected"), it does not suffice for just the bytes of each particular member to be valid—all of the pointer (or offset) linkages must also be valid. Should a link point to an invalid location, a method that uses that link to mutate its target object will likely mutate something else, lending itself to exploitation use (as demonstrated by a large amount of hacker research, e.g., [3, 23, 26] and the most recent [5]).

This poses a conundrum for the programmer writing such code units. To make sure that all the linkages are correct (for the semantics of data structure), one would potentially need to walk them all, which is unrealistic, inefficient, and would explode a simple task, such as adding or deleting an element from a heap free list or a hash table's collision list, into verifying a lot of things about the entire table first. In other words, the context needed to verify the data being worked on is too large and non-local. So this is not how such code is written. This brings us to how it's actually done.

A typical accompanying expectation for these code units is that they are substantially isolated from (properly validated) inputs—otherwise maliciously crafted input easily leads to numerous heap metadata and object store exploits that make exploit primitives out of innocent object maintenance routines. Therefore, when programmers write such code, they make believe that all the previous invocations of related code units have left the data units in valid linkage states, and *no other code units have touched it*. In other words, the programmer's expectations of input validity are not checked byte-for-byte, but rather are based on the isolation of the data and its orderly access only by specifically allowed code units working in concert with the data being read or written—that is, it is based on *exclusive access relationships of code and data units*.

Broadly speaking, programmers conceive both intent and trustworthiness of these code units in terms of data flows from and to these units. Although a simple input parser would know if the next byte is wrong because of the previous bytes, a data collection maintenance method such as a heap manager or hash table manager method cannot realistically visit and check all the bytes that matter for determination of validity. So the granularity level of programmer expectation is, "no other code has touched this heap segment".[19]

This expression also has an attractive relational duality[20] in its expression of intent semantics: intent of code is defined by the data it is expected to access, and vice versa. To the best of our knowledge, this approach has long been neglected (at least since the demise of segment-based programming models and tagged architectures), with the recent exception of *SegSlice* [11].

Coming back to our *gostak* example, an intent-level policy would play the role of the grammar to express the ubiquitous programmer expectations described above—expectations ubiquitously ignored at runtime. Only the labeled *gostak* may *distim* the labeled *doshes*, and it's only the *doshes* that are being acted on by the *gostak.* This is all we know, and it is enough for maintaining a level of programmer intent. All of these entities are opaque, as well as their intended actions, but we trust the whole so long as no other actors and actions are in the picture. A heap metadata chunk is only to be operated on by heap manager, and a heap manager is what operates on heap metadata chunk. If it *distims* the *doshes*, it's the *gostak*, and the *gostak* is what *distims* the *doshes.* Should *drokes* or *duskats* attempt to act on the *doshes*, we know something gets untrustworthy about the *doshery*.[21]

We can draw an analogy to what's called *duck-typing*: if it read/writes duck data, it's a duck, and that's all we know. Then we describe the duck code to the runtime reference monitor in these very reduction terms: enforce such relations, keep non-duck-labeled code from touching duck-labeled data, let only the marked *gostak distim* the labeled *doshes.*

## 7.3 Memory Flow Barriers between Code Sections

One important aspect of enforcing the exclusive relationships between code and data section is *isolating code from data that it is not expected or intended to consume.* Indeed, programmers assume such isolation when they first apply sanity checks and transformations to input data, and then write subsequent code under the assumption that the data now has been "cleared" and can be operated on safely. The attacker's potential ability to slip in other, uncleared data to code that expects cleared data likely breaks its explicit or implicit assumptions and leads to exploitation by maliciously crafted data payloads. Indeed, effective lack of such isolation underlies

[19] Note that hardened heaps such as Microsoft's Low Fragmentation Heap use canaries and XOR-ed check areas within the heap to create *local* validation contexts for heap manipulator code units that comprise the heap manager, as tell-tales of other code units' forbidden access; the security assumption there is that such an unexpected access will trip the tell-tale and will invalidate the few bytes close (and checkable) to the pointers or offsets being mutated by the manager code unit. Thus these hardened heaps bring granularity of checkable local context back towards the parsers' end of the spectrum.

[20] "Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious." (F.P. Brooks, *The Mythical Man-Month* [13])

[21] These additional entities appear in the interactive fiction "The Gostak" by Carl Muckenhoupt; it is freely available online at several interactive fiction sites.

most modern "no-code" exploitation scenarios for binary executable targets.

As an example, consider a parser that is intended to scan the input data and either "sanitize" or "normalize" it, or to reconstitute it and build trusted data structures for further processing. Once that parser is done, none of the raw data in its input buffers should be accessible by subsequent parts of the process if these are meant to consume only "sanitized" and reconstituted data. However, in multi-stage exploitation scenarios that rely on parser bugs for their initial stage(s), that raw input data contains payloads for further stages. These payloads are brought into play by triggering the initial parser bug; blocking the data flow from raw buffers to vulnerable code would break the execution of the exploit.

Such a "memory wall" was introduced by The UDEREF part of the GrSecurity's PaX[22] hardening patch for the Linux kernel. Empirically, this wall has dramatically reduced the incidence of privilege elevation attacks. The kernel programmers' clear intention is for the kernel code to access only such user memory contents that have been validated, e.g., passed the appropriate checks at the start of a system call. Lower layers of the system call's implementation rely on such checking to ensure that they only receive coherent and well-formed data structures. Consequently, exposing kernel code to data structures constructed by the attacker in userland led to exploits: kernel code executing at the highest privilege corrupted kernel memory while operating on the crafted userland data. UDEREF[23] prevented a large number of such unintended data flows by blocking the kernel from accessing most of userland memory pages; the result was a dramatic reduction in exploitability of the hardened kernel. Notably, the new SMAP feature of x86 processors opens a way to more efficient implementations of UDEREF.[24]

ELFbac applies the same principle to the ABI-granular, intention-level semantic division of a process' code and data units.

## 7.4 Implicit Semantics of Standard ELF

Binary executable formats have been designed to present to the OS a minimal description of the functionally different kinds of data (including compiled code) required to construct the process runtime image. These formats include special sections of metadata describing the extents and function of subsequent binary sections; these metadata sections serve as the instructions for the OS loader specifying the address ranges to copy the sections to, and the desired protections of the underlying memory pages.

Thus loader formats such as ELF already have implicit intent-level access semantics at the granularity of their code, data, and metadata sections. In ELF, these semantic relationships are partially expressed in the section header table fields (e.g., in ELF's *Info* and *Link* section entry fields), and partially implied by standard section name pairings (such as *.init* and *.ctors*, *.fini* and *.dtors*, *.plt* and *.got*, and others).

As the engineering of the Unix process' runtime became more sophisticated, ELF accommodated this trend by defining the necessary new metadata structures. In parallel, ELF added features to support more advanced debugging and exception-handling mechanisms. As a result of this co-evolution, a typical ELF executable binary contains over 30 sections, each of which corresponds to code and data units with different runtime semantics—expected behavior defined *in relation to other sections*.

This trend towards more detailed internal descriptions allowed ELF to unify the representation of binaries-to-be-loaded, and various intermediary stages of compilation and runtime. In particular, ELF is used to represent the compiler's object code files (".o"), shared libraries (a.k.a. shared objects, ".so"), and even core dump files.

ELF files are largely composed of ELF sections. An ELF section is simply a set of contiguous bytes in a ELF file and is described by an ELF section header, of which ELF maintains a table. Some ELF sections contain or describe the code and data of the program proper (`text` `rodata`, `data`). Others, such as `bss`, take no space in the file besides the entry in the section table. The `bss` section in particular describes the necessary allocation size for the program's uninitialized global variables. Other sections contain code and the necessary supporting data to be executed before the start of the main program or after its end (regular or forced) to set up or tear down the runtime. Examples of such sections are `ctors`, `dtors`, `init`, and `fini`, which control the calling of the constructor and destructor code for the objects in the runtime address space, including the necessary initialization of the runtime's own structures. (Here and later, we use the term *object* for a semantically distinct ELF code or data unit, rather than in pure OO or C++ ABI sense.)

The contemporary ELF design separates both the special purpose code and its data into separate ELF sections, named according to their semantics. This design decision creates a special and *exclusive* relationship between these sections. A violation of this relationship at runtime is likely the sign that the process should no longer be trusted. If the implicit read, write, and execute relationships between these ELF sections were enforced, a number of exploitation techniques that use the contents of these sections contrary to their intended use and place

[22] `http://pax.grsecurity.net/`, `http://pax.grsecurity.net/docs/`

[23] For the discussion of UDEREF's implementation details and efficacy, see `http://grsecurity.net/~spender/uderef.txt`

[24] See "Supervisor Mode Access Prevention for PaX", `http://forums.grsecurity.net/viewtopic.php?f=7&t=3046`

in the process timeline would become infeasible. For a case in point, the data contained in `ctors` and `dtors` is only meant to be interpreted and acted upon by `init` and `fini`, respectively. A deviation from this access pattern may be a sign of runtime loss of trustworthiness. For example, Bello Rivas [37] demonstrates a method of exploiting a GCC-compiled program by making an entry in `dtors` that points to malicious code located outside the `fini` section.

Although this decomposition may have been originally motivated by abstracting away the algorithmically common part and the program-specific data part of the simple setup or teardown ("loop over a table of function pointers to constructors/destructors, calling each one in order"), the contents of `.init` need not be limited to a single simple loop, nor the contents of `.ctors` to a simple function pointer table, and the same for `.fini` and `.dtors`. As long as the contents of these sections agree, they will be true to their ELF-semantic roles with a more complex algorithm and more complicated data structures to drive them. This same argument can be applied more generally to any set or related code and data units.

**PLT and GOT.** The `got` (global offset table) and `plt` (procedure linkage table) also show the special relationship between code and data ELF sections. The `plt` contains call stubs to external library functions to be linked at runtime by the dynamic linker, whereas the `got` contains the addresses through which these functions are indirectly called.

Initially indirectly calling a function through its `got` entry gives control to the dynamic linker which, with the help of this stub, is provided with an index of the function's dynamic linking symbol, including its name. After the dynamic linker resolves the symbol name and loads the appropriate libraries, it rewrites the `got` entry to point directly to the address where the function is loaded, so that further indirect calls through the `plt` result in calling that function with just a single indirection (through its `got` entry).

Although only `plt` is expected to read the `got` while dispatching through it, the dynamic linker is expected to write its entries. However, the rest of the code in the process' address space should not write `got` entries; in fact, any of its code unit doing so likely indicates a dynamic code hijacking [17, 43]. Notice also that any given `got` entry is meant to be written only once.

**Dynamic directory.** The dynamic sections, indexed in the `dynamic` section and segment, together form a complex data structure consisting of symbol tables, their accompanying string tables, version tables, hash tables, the `got` and the `plt`, the relocation data for all of the above as necessary, and a global directory to all of the above.

These semantics could provide security policies—and we can provide tools to help the developer extend these policies and make them explicit. However, the conventional "forgetful" Unix OS loader discards all this wealth, much to the attackers' delight.

## 7.5 Making the Loader Unforgetful of ELF Semantics

Unfortunately, under the current UNIX implementation conventions, the OS loader forgets all ELF information that went into creating the process' runtime virtual address space. We call this the **forgetful loader** problem. In particular, the OS loader is driven by the *segment* header table rather than by the section header table. Segments aggregate many semantically different sections based merely on their memory read/write/execute profiles, and ignore other semantic differences.

For example, in ELF, the `.text` section is often grouped with the `.rodata` section into the same *"R-X"* segment because both sections are expected to be non-writable (and an attempt to write them indicates untrustworthy behavior); however, this is just about the only thing they have in common, semantically. The one important exception to this is the two segments that serve dynamic linking: the short `.interp` segment that contains the path to the dynamic linker-loader, and the `.dynamic` segment that indexes all sections referenced by the dynamic linker in its runtime operation.

Thus the loader, as it is currently conceived of, is the weak link in the chain of passing semantic information on ELF units. Both the assembler and the linker support source code pragmas for placing generated code and data in custom target sections; the loader, however, operates only on section-aggregating segments, and ignores sections. This appears to be a decision influenced by the paucity of OS/kernel and MMU support for discriminating between different program semantics units at runtime.

However, technically there is nothing preventing the OS loader and kernel from being as aware of ELF sections as the components of the development toolchain or the dynamic linker: the section header table is either typically mapped into the runtime address space, or can be trivially mapped so. Should the sections be relocated, this table could also easily be patched by means of the standard relocator implementation.

In fact, crucial section information is replicated in the dynamic symbol tables accessible to the dynamic linker through the `dynamic` segment, and is so made available in its context. This decision probably reflects the conceptual gap between the legacy, limited *OS loader/kernel*'s view of the runtime process semantic structure, and the more recent *dynamic linker/loader*'s view of this structure (which the latter manages).

However, these is no reason why we should stick to

this disparity between an older and a newer design and keep the OS loader to a dumbed-down and forgetful view. Our vision is outlined in the next section.

# 8 Prototypes

We implemented software ELFbac prototypes for Linux on x86_64 and ARM. However, our design aims for portability across operating systems and ABI architectures that generally follow the COFF format design, such as Microsoft's PE or Apple's Mach-O.

Our implementation is based on existing virtual memory capabilities in x86 and ARM processors. We create a separate virtual memory context for each policy state, so that memory accesses relevant to our policy cause page faults, which we intercept in the kernel. Just as threading adds a one-to-many relationship of memory contexts to execution flows, we create a many-to-many relationship—since each execution flow can now change its memory context with each FSM state transition.

## 8.1 Implementation

We produced the original prototype for x86_64 and later ported it to ARM, to add an extra layer of policy to industrial control embedded systems we were hardening. We describe the ARM-specific details after the overall description of the x86_64-based prototype.

**Per-state shadow memory contexts.** Our prototypes replace the kernel's view of a process' virtual memory context with a collection of "shadow" contexts, one per each policy FSM state. Each shadow context only maps those regions of memory that can be accessed in the current state without violating the policy (which corresponds to loops in FSM). Any other access is trapped; the trap either causes a transition to another allowed FSM policy state, or is a policy violation.[25] On a legitimate state transition, the virtual memory context is changed; on a violation, appropriate handling logic (e.g., logging of the violation event and process termination) is invoked.

These state-specific memory contexts are only used by the page fault handler and in context switching routines. For the rest of the kernel functionality, the original virtual memory context still serves as a global view of the process address space (of which each of the shadow contexts makes a subset accessible). Therefore, we do not need to change any other subsystems in the kernel or any userspace code, as they can continue to use the system calls and kernel APIs to modify the address space, without impacting ELFbac enforcement.

[25]This implementation makes it hard to implement "once-only" access of a memory region as a policy primitive; we will pursue this in future work.

The state-specific shadow contexts are lazily filled—every state starts empty and is filled by the page fault handler. Initially, an empty shadow virtual memory context is created for every state in the policy and the context corresponding to the initial state is activated. Whenever the process accesses an address not mapped in this context, the MMU raises a page fault, which ELFbac intercepts. If the access is allowed by the policy and does not result in a state transition, the corresponding memory mappings (and page table entries) are copied to the shadow context, so they will not cause a fault when accessed in the future from the same state. If, on the other hand, the access results in a state transition, the shadow virtual memory context corresponding to the new state is activated.

Whenever a region in the original memory context is changed, it is unmapped in all the shadow context, so the next access to this area will again be validated against the policy.

**Additions to the process descriptor.** Our implementation adds only three pointers to the Linux kernel task structure (`task_struct`).

First, a pointer (`elf_policy_mm`) to a per-state separate virtual memory context is added. As explained above, this pointer is only used by the page fault handler and in context switching routines, so that the remaining large portion of the kernel, which accesses the traditional virtual memory context pointer (`tsk->mm` in Linux) to manipulate mappings, address holes or for accounting purposes, retains the illusion of a single address space per process. This reference also makes sure that the same address is never re-used and allows familiar debugging tools to work unmodified with the `ptrace` system call.

Secondly, a pointer (`elfp_current`) to the policy currently in effect for a process is stored. The policy is stored as a digraph of states, where each node stores a binary search tree of edges to other states.

Thirdly, a stack of userspace stack pointers is stored in the `elfp_stack` pointer, which is used to implement the stack-changing mechanism.

**Per-state shadow spaces as a new caching layer.** In theory, the policy is verified on every memory access by hooking the page fault handler. As trapping every memory access would be catastrophic for performance, we add a layer of "caching" on top of the existing hardware caching layers.

Consider a memory access policy in standard UNIX. The authoritative view of memory is given by the `mm_struct` struct of a process. The actual page tables (pointed to by `CR3` on x86) serve as a "just-in-time compiled" expression of the policy, since the appropriate PDE/PTE entries are created as needed by `mmap` and other system calls. Thus, the page table serves as a "caching layer" for the actual policy expressed in

vma_structs. Moreover, page table look-ups are cached by the TLB structures (separate for instruction and data paths on x86).

We generalize this architecture by adding another layer of shadow memory for each process, within which different code sections (corresponding to the phases of the process or policy FSM states) receive different views of memory via different page tables—and must therefore generalize the caching accordingly. Namely, the shadow contexts create an extra load on the TLBs, as it would cause them to be flushed on every FSM state change. To somewhat reduce this impact, we use our "lazy" design for filling per-state shadow contexts, in which each policy state starts with an empty page table and gets the appropriate mappings copied into each on each valid access. This design emulates the relationship of between the page tables and the TLB: just like the TLB, the policy state's page table accumulates successfully accessed mappings.

As soon as the addresses are unmapped in the global context, they are deleted from each page table as well. The page table associated with the current state is loaded on each context switch.

On newer architectures, we additionally reduce the performance impact of shadow contexts by using PCID tags.

As an architectural desiderata, we posit that since caching is heavily involved on the path of enforcing a memory access policy, it should in fact become an *actor* in enforcing this policy. This means, in particular, that cache entry invalidation must be elevated from the status of a heuristic to that of a policy primitive. Our full technical report includes a proposal for *FlexTrap*, a design that connects caching and trapping policy primitives.

**Per-state syscall checking.** In addition to the above virtual memory manipulation, the current state—i.e. the label of the code unit (ELFbac section) currently executing—is verified on every syscall. Each policy state either allows or disallows each syscall; more fine-grained checking can be delegated to userspace wrapper functions.

**Policy entry points and loading.** All Unix-family processes (after `init`) are originally created through a `fork()`-type syscall. This syscall preserves the ELF policy of the parent process.[26] When a new executable is loaded with the *exec()* call, the policy is replaced with the policy of the new executable.

On dynamic binaries, we use the linker to set up the policy. If the ELF binary is dynamically linked, the kernel first loads a statically linked interpreter binary[27], whose policy will be loaded initially. The main executable is then relocated and set up by the interpreter. Since the ELFbac policy is just another ELF object, the linker is already equipped to relocate the policy as needed. Before control transfers to the main program, the interpreter calls a special system call that loads the relocated policy. Thereafter, this system call is disabled until a new binary is set up. Hence, neither the policy parser nor any other part of the kernel have to worry about how the address space was set up, e.g. by ASLR.

## 8.2 ELFbac on ARM

We used ELFbac as hardening layer for a control application running on an industrial (embedded) computer. The application involved parsing a complex ICS protocol (DNP3) and selectively proxying its "safe" subset. Consequently, the policies we deployed in this system were much simpler—essentially, isolating the raw input buffers from any code but the validating parser's, and the parser from any application data other than the well-type protocol objects it created. However, the control application required stricter performance bounds, which had to be measured for all the relevant state transitions. We outline these measurements in Section 9.3.

Here is the summary of the changes we needed for the port.

Rather than keeping a complete `mm_struct` per ELFbac state, we instead tracked just a separate page table per state, reducing the amount of synchronization needed between, e.g., the VMAs in between ELFbac states on various `mm` events. We defined a new format for binary policies consisting solely of fixed-width integers defining the relevant policy objects, simplifying parsing of binary policies in the kernel. We loaded ELFbac policy on an ELF file load from `binfmt_elf` rather than by syscall. We optimized support for userspace stack switching. Finally, we used ARM ASIDs to reduce cost of ELFbac transitions.

In summary, our porting experience validated our design decisions and the overall ELFbac architecture.

# 9 Evaluation

There is a difficulty in estimating the performance impact of ELFbac, which is inherent in the measurement of any security primitive: it ultimately depends on the software it is used to protect how often and on what code paths it is going to be hit. Our hypothesis is that well-designed software already exhibits sufficient modularity

[26] If a fork should be treated as a special policy event, the *fork()* library function can be wrapped to provide the transition just after the fork. Access to the *fork()* system call from any other policy state should then be prohibited.

[27] On Linux, this is usually `/lib/ld.so`.

and locality to make frequent (and costly) ELFbac context switching unnecessary, and amortizing the cost of these switches over the productive sections and states of the program.

Our performance measurements for the x86_64 and ARM were structured differently, due to the difference in the respective policy applications. For x86, we tested popular server and library software in both a Qemu-based virtualized environment (with an eye towards cloud applications) and on bare AMD and Intel hardware. Our goal was to roughly evaluate the overall overhead for such existing software.

For our ARM platform project, our policy was a part of a hardened industrial control system (ICS), dedicated to a specific control application with stricter performance needs. Thus we focused on micro-benchmarking the relevant state transitions to make sure the policy did not degrade the application response.

## 9.1 Security: implicit flows eliminated

The primary advantage of protecting systems with ELFbac is lessening the impact of an attacker. By preserving and enforcing intent-level semantics, an attacker who has gained code execution or an information leak in one part of the application can no longer compromise other parts. Among other benefits, ELFbac makes attack techniques like return-oriented programming much more difficult.

In order to assess the efficacy of our system, we sampled four recent high-impact vulnerabilities from the Common Vulnerabilities and Exposures database[28] that could result in code execution.Because either the software was not available in the first place or adequate vulnerability details were not disclosed, we did not attempt to write an actual ELFbac policy for the relevant software. Instead, we consider how a sensible default ELFbac policy, such as generated by our preliminary tools, would limit the exposure from each vulnerability.

**CVE-2012-3455.** In this vulnerability, a heap buffer overflow in the Word processor permits an attacker to craft evil input that subverts the main KOffice process.

ELFbac could be used to separate the Word parser and the main KOffice process. If an attacker crafts a document that can perform computation with the parser, he will no longer be able to read arbitrary files or spawn a shell, but limited to using the internal KOffice API.

**CVE-2012-3639, CVE-2012-2334.** In CVE-2012-3639, a bug in WebKit allows an adversary to subvert an entire browser; in CVE-2012-2334, a bug in an image library allows an adversary to subvert an entire office suite.

In both vulnerabilities, an application uses a parser for display formats (HTML and PNG, respectively). However, both parsers only require minimal interaction with the remaining components of a program. Other browsers such as Google's Chrome [9] already demonstrate that WebKit only requires a handful of calls to render bitmaps and consume input events. A PNG parser should only be able to read an input buffer and write an output buffer. Neither require access to any system calls or other program data structures.

ELFbac could mitigate both vulnerabilities by confining the parser to exactly what it needs to touch, thus preventing a compromised parser from subverting the rest of the application. As opposed to the Chrome security model, ELFbac does not require re-engineering of existing code and the same policy mechanism can be ported across different operating systems.

**CVE-2012-0199.** In this vulnerability, a bug in the processing of SQL database queries lets the attacker subvert the rest of the application.

Usually, SQL injection attacks result from input text fragments being copied into a SQL query without proper escaping. To mitigate this, various database frameworks, such as Microsoft .NET, use dedicated query builder mechanisms. With ELFbac we can assure that no code but the query builder can send data to the SQL client API, defending both against inadvertent SQL injection vulnerabilities and against the SQL layer being used after another exploitation.

## 9.2 Stability and Performance: x86_64

In this section we explain our evaluation criteria and present a sketch of results. Cross-cutting performance evaluation of key Linux software is still ongoing.

**ELFbac policy injection: stability and coverage.** ELFbac requires a policy section to be injected into ELF binaries. In order to gain acceptance, such injection should be seamless and work for the absolute majority of ELF-based software.

Accordingly, we decided to target an entire GNU/Linux distribution in our testing. We chose the long-term support Ubuntu 12.04 64-bit distribution, and took advantage of its packaging system to locate, rewrite, and test the resulting rewritten software packages.

For this test, we deployed the publicly available ELF-rewriting tool *Mithril*[29] on the Amazon S3 cloud, using PiCloud as the front end to access the cloud and drive the experiment. The rewriting involved about 45,000 Ubuntu packages in a roughly 20 GB repository. We successfully rewrote the ELF binaries in the distribution's packages, inserting the `.elfbac` policy section and transforming the file structure to a "canonical" form defined by a DSL implemented by the tool.

[28] http://cve.mitre.org

[29] `http://github.com/jbangert/mithril`

The rewriting took 260 core hours for two rebuilds. We encountered a total of 20 build failures, which we discovered to have occurred in `*-data` packages, containing non-binaries. We then functionally tested the resulting distributions using `debootstrap`, with no failures.

**Overall performance hit.** We are still searching for a methodology that would allow us to separate ELFbac's inherent costs from the incidental ones easily avoided by a programmer who decides to use it. Thus the following case studies to estimate ELFbac's performance hit are preliminary and likely overestimate ELFbac's costs.

In a nutshell, ELFbac overhead measured on an emulated system (Linux `3.4/kvm+3.12`) was approximately **9–10%**. On bare hardware, we encountered a drastic difference between performance on our AMD (Opteron 6320) and Intel (Xeon E3-1245, Sandy Bridge) test systems. Specifically, whereas on AMD our overhead was approximately around **3%**, on our Intel system in some cases without our hardware optimizations it approached 25%. By using hardware optimizations of virtual memory translation such as Intel's recent PCID feature, we managed to somewhat reduce this overhead (e.g., resulting in an approximately 85% reduction of DTLB misses), but such gains remain load case-specific and require more comprehensive measurement.

We attribute this difference to the respective TLBs flushing patterns when switching between multiple address spaces. Below we report the *worst* cases of performance, despite our AMD overheads being consistently lower.

**Libpng benchmarks.** To evaluate the performance of our prototype, we wrote an ELFbac policy for `libpng` and benchmarked image decoding performance. This mirrors the core use-case of ELFbac to isolate large, untrusted legacy code modules that perform CPU-heavy computation on relatively well-defined inputs; `libpng` is representative of such with its rich API of over 450 functions and callbacks.

Our test is based on publicly available benchmark code[30], modified to decode an image, perform minimal processing (flip two color channels) and re-compress the image 100 times, applied to Wikipedia's PNG sample. The benchmark exercises 39 libpng calls, including one call per row of pixels. The case is "medium-bad", as it is CPU-bound and has many inter-module function calls but no no complex memory operations such as `mmap()/munmap()`.

We evaluated our prototype both on the KVM hypervisor on a 3rd generation Core i7 processor as well as on bare hardware:

[30] `http://zarb.org/~gc/html/libpng.html`

| *Platform* | *Avg. sec.* | *Avg. sec. with ELFbac* |
|---|---|---|
| KVM + i7 | 1.06 | 1.16 |
| Physical i7 | 1.01 | 1.32 |

**Nginx benchmarks.** We tested the popular Nginx web server, a known high-value target for attackers. Nginx input-parsing code is a complex state machine designed from scratch, but Nginx still depends on several libraries: *PCRE*, *OpenSSL*, and *Zlib*, as well as on `libc` for its system interactions. We tested the ELFbac policy of separating these libraries from the main body of Nginx.

We used the *ApacheBench* as our performance benchmark. As before, performance overhead under KVM-emulated was smaller than on bare hardware: 9-15% on the former, and up to 30% on the latter. We note that the poor results on bare hardware are clearly due to too many state transitions on the hot path; whereas KVM itself incurs performance losses, and optimizes virtual memory handling against those.

These case studies show that ELFbac will stand to profit from further CPU virtualization performance enhancements such as tagged TLBs, but also may already be reasonable to use with high-assurance virtually hosted applications—such as on cloud platforms.

## 9.3 Performance on ARM: micro-benchmarks

We performed the following tests on the Altera SoC FPGA development board that was the basis of our targeted industrial computer.

We also considered testing performance under Qemu ARM emulation, to compare the overheads on "bare hardware" vs emulation. However, we discovered that the Qemu ARM support doesn't include emulating ASIDs and always performs a full TLB flush, making the apparent performace under emulation appear much worse than on actual hardware.

For each of the tests below, a description of the benchmarked state transition path and the average `time(1)` times for each test are reported. The microbenchmarks' code and raw outputs are included in our GitHub sources. The *nopol* cases are those with ELFbac disabled, *pol* enabled.

Basic state transition between libc and the app: `start->libc->start->libc->start->libc (exit)` performing IO along the way (`simple_pol` vs `simple_nopol`):

| | | | |
|---|---|---|---|
| nopol | real | total: 0.0 | avg: 0.0 |
| pol | real | total: 0.01 | avg: 0.001 |
| nopol | sys | total: 0.0 | avg: 0.0 |
| pol | sys | total: 0.0 | avg: 0.0 |
| nopol | user | total: 0.0 | avg: 0.0 |
| pol | user | total: 0.0 | avg: 0.0 |

Thrash: transition `start->read->start->write several times then exit` (`thrash_pol` vs `nopol`)

nopol real total: 1.96 avg: 0.196
pol real total: 1.96 avg: 0.196
nopol sys total: 0.0 avg: 0.0
pol sys total: 0.0 avg: 0.0
nopol user total: 0.0 avg: 0.0
pol user total: 0.0 avg: 0.0

Sum: transition `start->heavy computation state->start->exit` (`sum_pol` vs `nopol`)

nopol real total: 483.5 avg: 48.35
pol real total: 483.5 avg: 48.35
nopol sys total: 0.0 avg: 0.0
pol sys total: 0.0 avg: 0.0
nopol user total: 483.4 avg: 48.34
pol user total: 483.4 avg: 48.34

Sum Thrash: transition `start->heavy computation state and back several times, then exit` (`sum_thrash_pol` vs `nopol`)

nopol real total: 241.81 avg: 24.181
pol real total: 241.81 avg: 24.181
nopol sys total: 0.0 avg: 0.0
pol sys total: 0.0 avg: 0.0
nopol user total: 241.8 avg: 24.18
pol user total: 241.8 avg: 24.18

In summary, most of the transition micro-benchmarks showed negligible overhead; those that didn't still remained under the responsiveness constraints of the application.

## 10 Related Work

Although the main thrust of ELFbac is to suggest a policy design that taps a previously underused source of information about programmer intent, its implementation can be considered a variant of Software Fault Isolation (SFI). Thus we discuss where our prototype fits in the SFI world, and then briefly comment on the relationship between ELFbac and control-flow integrity (CFI) mitigations.

### 10.1 SFI and ELFbac

SFI systems isolate the components of an application. The first modern system was introduced by Morissett [27] and improved by many authors [20, 42]. SFI is deployed in commercial web browsers with Native Client [44]. These systems all come at significant performance cost and allow only loose coupling between components.

Every SFI component has its own memory region and accesses across these regions are typically not supported. This requires the introduction of Remote Procedure Call style interfaces, which requires the programmer to modify the application and causes serialization overhead. Some SFI systems, such as Chrome with NaCL, therefore provide for only one isolated module, which is prevented from accessing the rest of the application, which in turn can access everything. This approach fails if we want to separate multiple components without a strict trust hierarchy. For example, we would like vulnerabilities in a SSL library to not affect web server data structures, while also preventing an exploited web server from leaking the keys stored by the SSL library.

Another problem with SFI is that a module always has the same set of permissions. However, in real systems modules are re-used for different purposes. A compression library might be used to compress responses sent to different users, or to compress log data. Each instance of the compression library should be isolated.

Newer systems such as BGI [16] and LXFI [25] address these challenges with capability-style systems. BGI enforces revokable byte-level permissions on modules, which can be used as capabilities to give different permissions to different invocations of a module. LXFI uses object-style capabilities and module *principals* to isolate different instances of the same module. Additionally, LXFI enforces data structure integrity, allowing memory to be shared between modules.

We improve upon these systems in the following key ways: temporal isolation, hardware-assisted enforcement, and variable granularity.

First, existing systems focus on isolating along existing composition boundaries provided by the programming environment, such as libraries and modules. ELFbac allows a domain to have different permissions during different phases of execution, e.g. during initialization and processing.

Second, ELFbac enforces policies via Linux's virtual memory system, as opposed to code rewriting. The use of the VM system, due to developments in CPU design, has performance potential equal or better to that of rewriting techniques, while improving flexibility and simplicity. Finally, we allow protection domains to be specified at different levels of granularity. Some aspects of a policy might be better specified as individual libraries, modules or even functions interacting, whereas in other cases the natural and convenient isolation boundaries might be a set of tightly coupled functions and data structures spanning across different libraries.

We note that using ELF data structures to infer enforceable program properties related to code structure is not limited to SFI, but can also help inform control flow integrity (CFI) measures. For example, Payer et al. [33] used import and export information from shared objects (`dynsym` and `symtab` if available) to approximate the programmer's intended control flow and enforce it via a CFI mechanism.

## 10.2 CFI and ELFbac

Control-flow integrity (CFI) is a stateless, mitigation-based approach to preventing exploitation. CFI uses static analysis to derive restrictions on the control flow path of a program, and then enforces these restrictions at runtime via a variety of mechanisms, killing the process instance on a violation.

A wide variety of CFI analyses and mechanisms have been proposed. Reviewing these is beyond the scope of this paper. Reviews and effectiveness studies can be found, e.g., in [2] and the recent [15].

We distinguish ELFbac's goals from those of the majority of CFI by noting that CFI is essentially a mitigation approach, *deriving* the programmer's intent from code, but providing no mechanism for the programmer to *explicitly specify* such intent as a concise, easily readable policy.

By contrast, ELFbac's goal is to provide the programmer with the succinct, explicit means of describing the functional partitioning of the program into code and data units, and by describing the units' access relationships, at the same granularity as these units are defined.

Thus ELFbac can complement the finer-grained CFI measures, or be a policy largely orthogonal to the CFI enforcement within the units.

# 11 Conclusion

Prior work in application security policies put much emphasis on explicitly restricting a program's access to OS resources and services, such as files and system calls, and on enforcing inferred, implicit developer expectations of the program's execution (including its control and data flow) based on the programming language abstractions. We argue that modern software composition also requires explicitly restricting the interactions of components within a process space: intra-process, inter-component access control, which expresses the rough intent of each component via its accesses—and show that leveraging linking information serendipitously allows the C/C++ developer to specify such an explicit policy without a cognitive overload.

Our prototype ELFbac allows programmers to specify and enforce intent-level semantics of different units of code and data, by reusing existing ELF ABI infrastructure. ELFbac maintains compatibility with legacy code and binaries, and noticeably raises the barrier to successful exploitation.

# Acknowledgments

This work was supported in part by the Intel Lab's University Research Office, by the TCIPG project from the Department of Energy (under grant DE-OE0000097), and by the Schweitzer Engineering Laboratories Inc., who made the ARM port of ELFbac for ICS systems possible.

We would like to particularly thank Felix 'FX' Lindner and other researchers at Recurity Labs for many productive discussions and critical insights. We gratefully acknowledge the design inspirations by the Grsecurity/-PaX project and the PaX team. On the subject of the ELF structure and ABI composition mechanisms, we owe many insights to the ERESI project by Julien Vanegue and the ERESI team, as well as to the Grugq's papers on the subject, and to the early works by Silvio Cesare, Klog, and others published in the *Phrack* magazine. We thank Rodrigo Branco for many helpful discussions of Linux kernel security.

Views expressed are those of the authors alone.

# References

[1] XFI. https://pdos.csail.mit.edu/6.828/2007/lec/l-xfi.html.

[2] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow Integrity Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur. 13*, 1 (Nov. 2009), 4:1–4:40.

[3] ANONYMOUS AUTHOR. Once upon a free(). Phrack 57:9. `http://phrack.org/issues.html?issue=57&id=9`.

[4] ARGP, AND HUKU. Pseudomonarchia jemallocum. Phrack 68:10, April 2012. `http://phrack.org/issues.html?issue=68&id=10`.

[5] ARGYROUDIS, P., AND KARAMITAS, C. Heap Exploitation Abstraction by Example. OWASP AppSec Research, August 2012.

[6] AVONDS, N., STRACKX, R., AGTEN, P., AND PIESSENS, F. Salus: Non-Hierarchical Memory Access Rights to Enforce the Principle of Least Privilege. In *Security and Privacy in Communication Networks (SecureComm)* (2013).

[7] BANGERT, J., AND BRATUS, S. ELF eccentricities. CONFidence 2013, Krakow, Poland, 2013.

[8] BANGERT, J., BRATUS, S., SHAPIRO, R., AND SMITH, S. W. The Page-Fault Weird Machine: Lessons in Instruction-less Computation. In *7th USENIX Workshop of Offensive Technologies (WOOT)* (August 2013). `https://www.usenix.org/system/files/conference/woot13/woot13-bangert.pdf`.

[9] BARTH, A., JACKSON, C., AND REIS, C. The security architecture of the chromium browser. Tech. rep., Stanford, 2008.

[10] BOSMAN, E., AND BOS, H. Framing Signals—A Return to Portable Shellcode. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (2014), pp. 243–258.

[11] BRATUS, S., LOCASTO, M., AND SCHULTE, B. Segslice: Towards a new class of secure programming primitives for trustworthy platforms. In *Trust and Trustworthy Computing*, A. Acquisti, S. Smith, and A.-R. Sadeghi, Eds., vol. 6101 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2010, pp. 228–245.

[12] BRATUS, S., LOCASTO, M. E., RAMASWAMY, A., AND SMITH, S. W. New Directions for Hardware-assisted Trusted Computing Policies (Position Paper). In *Future of Trust in Computing*, D. Gawrock, H. Reimer, A.-R. Sadeghi, and C. Vishik, Eds. Vieweg+Teubner, 2009, pp. 30–37.

[13] BROOKS, F. P. *The Mythical Man-Month*. Addison-Wesley, 1975, 1995.

[14] CARDELLI, L., AND WEGNER, P. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys 17*, 4 (January 1985), 471522.

[15] CARLINI, N., BARRESI, A., PAYER, M., WAGNER, D., AND GROSS, T. R. Control-flow Bending: On the Effectiveness of Control-flow Integrity. In *Proceedings of the 24th USENIX Conference on Security Symposium* (2015), SEC'15, USENIX Association, pp. 161–176.

[16] CASTRO, M., COSTA, M., MARTIN, J.-P., PEINADO, M., AKRITIDIS, P., DONNELLY, A., BARHAM, P., AND BLACK, R. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 45–58.

[17] CESARE, S. Shared Library Call Redirection via ELF PLT Infection. Phrack 56:7. `http://phrack.org/issues.html?issue=56&id=7`.

[18] DULLIEN, T. Exploitation and state machines: Programming the "weird machine", revisited. In *Infiltrate Conference* (Apr 2011).

[19] ERIK BUCHANAN AND RYAN ROEMER AND HOVAV SHACHAM AND STEFAN SAVAGE. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proceedings of the 15th ACM conference on Computer and Communications Security* (2008), pp. 27–38.

[20] ERLINGSSON, U., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. Xfi: Software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), USENIX Association, pp. 75–88.

[21] FORD, B., AND COX, R. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX Annual Technical Conference* (Boston, MA, 2008), USENIX, pp. 293–306.

[22] JAY FREEMAN (SAURIK). Exploit & Fix Android "Master Key"; Android Bug Superior to Master Key; Yet Another Android Master Key Bug. `http://www.saurik.com/id/17,18,19`, 2013.

[23] JP. Advanced Doug Lea's malloc exploits. Phrack 61:6. `http://phrack.org/issues.html?issue=61&id=6`.

[24] KAMINSKY, D., PATTERSON, M. L., AND SASSAMAN, L. PKI Layer Cake: New Collision Attacks against the Global X.509 Infrastructure. In *Financial Cryptography* (2010), Springer, pp. 289–303.

[25] MAO, Y., CHEN, H., ZHOU, D., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Software fault isolation with API integrity and multi-principal modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 115–128.

[26] MAXX. Vudo malloc tricks. Phrack 57:8. `http://phrack.org/issues.html?issue=57&id=8`.

[27] MCCAMANT, S., AND MORRISETT, G. Evaluating SFI for a CISC architecture. In *Usenix Security* (2006), vol. 6.

[28] MÜLLER, T. ASLR smack and laugh reference. In *Seminar on Advanced Exploitation Techniques*. RWTH Aachen, February 2007.

[29] NERGAL. Advanced return-into-lib(c) exploits: the PaX case study. Phrack 58:4. `http://phrack.org/issues.html?issue=58&id=4`.

[30] NICOLAS CARLINI AND ANTONIO BARRESI AND MATHIAS PAYER AND DAVID WAGNER AND THOMAS R. GROSS. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security 15)* (2015).

[31] NOORMAN, J., AGTEN, P., DANIELS, W., STRACKX, R., HERREWEGE, A. V., HUYGENS, C., PRENEEL, B., VERBAUWHEDE, I., AND PIESSENS, F. Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base. In *22nd USENIX Security Symposium* (2013), pp. 479–498.

[32] OAKLEY, J., AND BRATUS, S. Exploiting the hard-working dwarf: Trojan and exploit techniques with no native executable code. In *WOOT* (2011), pp. 91–102.

[33] PAYER, M., BARRESI, A., AND GROSS, T. R. Fine-Grained Control-Flow Integrity Through Binary Hardening. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 12th International Conference, DIMVA 2015, Milan, Italy, July 9–10, 2015* (2015), M. Almgren, V. Gulisano, and F. Maggi, Eds., Springer, pp. 144–164.

[34] REBECCA 'BX' SHAPIRO AND SERGEY BRATUS AND SEAN W. SMITH. Weird machines in ELF: a spotlight on the underappreciated metadata. In *Proceedings of the 7th USENIX conference on Offensive Technologies (WOOT* (2013).

[35] RICE, H. G. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society 74*, 2 (1953), pp. 358–366.

[36] RIQ, AND GERA. Advances in format string exploitation. Phrack 59:7. `http://phrack.org/issues.html?issue=59&id=7`.

[37] RIVAS, J. M. B. Overwriting the .dtors section. `http://packetstormsecurity.org/files/23815/dtors.txt.html`, December 2000.

[38] ROEMER, R., BUCHANAN, E., SHACHAM, H., AND SAVAGE, S. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur. 15*, 1 (Mar. 2012), 2:1–2:34.

[39] SCHUSTER, F., TENDYCK, T., LIEBCHEN, C., DAVI, L., SADEGHI, A.-R., AND HOLZ, T. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (2015), pp. 745–762.

[40] SERGEY BRATUS AND MICHAEL E. LOCASTO AND MEREDITH L. PATTERSON AND LEN SASSAMAN AND ANNA SHUBINA. Exploit Programming: From Buffer Overflows to "Weird Machines" and Theory of Computation. *;login: 36*, 6 (December 2011).

[41] SKAPE. Locreate: An Anagram for Relocate. *Uninformed 6* (2007).

[42] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. In *ACM SIGOPS Operating Systems Review* (2003), vol. 37, ACM, pp. 207–222.

[43] THE GRUGQ. Cheating the ELF: Subversive Dynamic Linking to Libraries .

[44] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy* (2009), IEEE, pp. 79–93.

[45] ZHOU, Y., WANG, X., CHEN, Y., AND WANG, Z. ARMlock: Hardware-based fault isolation for ARM. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 558–569.

## Appendix A: ELFBac Kernel Structures

```
/* These structures describe an ELFbac policy in kernel memory. They
   are created at runtime from the elfp_desc structures found in the
   .elfbac section. This header file is intended to be used in all
   ELFbac ports, so per-kernel aliases from elfbac-linux.h are used.*/

struct elf_policy{
  struct elfp_state *states;
  struct elfp_stack *stacks;
  elfp_atomic_ctr_t refs; /*should be made atomic_t */
};
struct elfp_state {
  elfp_context_t *context; /* This memory context maps a subset of the
                              processes tables and is filled on demand */
  elfp_tree_root calls; /*Maps to OS tree implementation*/
  elfp_tree_root data;
  struct elfp_state *prev,*next; /* Linked list of states */
  struct elf_policy *policy; /* Policy this state belongs to */
  struct elfp_stack *stack; /* Last call transition taken */
  elfp_id_t id; /* id of this state in the policy. Used for parsing
                   policy statements */
};
struct elfp_stack {
  struct elfp_stack *prev,*next;
  elfp_id_t id;
  uintptr_t low,high;
  elfp_os_stack os;
};
struct elfp_call_transition{
  elfp_tree_node tree; /* Wraps OS rb-tree implementation*/
  struct elfp_state *from,*to;
  uintptr_t offset; /* Called address */
  short parambytes; /* bytes copied from caller to callee*/
  short returnbytes; /* bytes copied from callee to caller. <0 to
                              disallow implicit return */
};
struct elfp_data_transition {
  elfp_tree_node tree;
  struct elfp_state *from,*to;
  uintptr_t low, high;
  unsigned short type; /* READ / WRITE flags */
};
```