SELF: a Transparent Security Extension
for ELF Binaries [*]

Daniel C. DuVarney
dand@cs.sunysb.edu

Sandeep Bhatkar
sbhatkar@cs.sunysb.edu

V.N. Venkatakrishnan
venkat@cs.sunysb.edu

Department of Computer Science
Stony Brook University
NY, 11794

ABSTRACT

The ability to analyze and modify binaries is often very useful from a security viewpoint. Security operations one would like to perform on binaries include the ability to extract models of program behavior and insert inline reference monitors. Unfortunately, the existing manner in which binary code is packaged prevents even the simplest of analyses, such as distinguishing code from data, from succeeding 100 percent of the time. In this paper, we propose SELF, a security-enhanced ELF (Executable and Linking Format), which is simply ELF with an extra section added. The extra section contains information about (among other things) the address, size, and alignment requirements of each code and static data item in the program. This information is somewhat similar to traditional debugging information, but contains additional information specifically needed for binary analysis. It is also smaller, compatible with optimization, and less likely to facilitate reverse engineering, which we believe makes it practical for use with commercial software products. SELF approach has three key benefits. First, the information for the extra section is easy for compilers to provide, so little work is required on behalf of compiler vendors. Second, the extra section is ignored by default, so SELF binaries will run perfectly on all systems, including ones not interested in leveraging the extra information. Third, the extra section provides sufficient information to perform many security-related operations on the binary code. We believe SELF to be a practical approach, allowing many security analyses to be performed while not requiring major changes to the existing compiler infrastructure. An application example of the utility of SELF to perform *address obfuscation* (in which the addresses of all code and data items are randomized to defeat memory-error exploits) is presented.

1. INTRODUCTION

Attacks which exploit programming errors, such as buffer overflow and format-string attacks, are one of today's most serious security threats. Security advisories from the CERT coordination center indicate that such exploits constitute a majority of the attacks on the Internet, and the number of these attacks continues to grow rapidly. Several techniques have been developed to address the problem of finding vulnerabilities in systems and ensuring the safe execution thereof. We briefly describe the most common techniques below:

- **Run time monitoring.** In this approach, whenever an operation is performed, an interceptor is invoked which decides whether the particular operation is safe or not. The interceptor looks at the run-time state of the system and blocks any unsafe operation. Examples of this include modern intrusion detection systems, the Java runtime system, inline reference monitors, and so on.

- **Static analysis.** The program of interest is statically analyzed by a system which checks the program for security vulnerabilities before it is executed. Once certified by the analyzer, the code is usually not subjected to run-time checking, and is guaranteed to be safe. Proof-carrying code [30] and the JVM byte-code verifier adopt this approach to verify memory-safety properties.

- **Program transformation.** In this approach, an application to be executed is transformed into a program which has inlined security checks. Naccio [16] and SASI [14] use code transformation to ensure security.

All these techniques have been used successfully to ensure secure execution of programs. However, all of them require access to the program's source code, with the lone exception of SASI, which is dependent on the code-generation strategy of a particular compiler (gcc). Having access to source code is not practical for consumers of commercial applications outside of the open-source community, yet these are the largest group of users, and hence any security paradigm which ignores them is bound to fail. What is needed is the ability to take the security techniques mentioned above and empower users to apply them directly to binary code in a turnkey fashion. Furthermore, the new approach must not overburden producers of code with a host of restrictions and/or difficult tasks which must be performed, or it is unlikely to be adopted. Finally, the new approach must be sufficiently powerful enough to provide the ability to perform many useful security-related operations, such as the insertion of monitor code.

Unfortunately, existing binary formats offer very little support for analysis (and modification) for security vulnerabilities. Although

binary files typically have less program structural information than source programs, this does not preclude binaries from being successfully analyzed for security vulnerabilities. For any kind of analysis or transformation on binaries, it is important to be able to retrieve information from binaries and also to manipulate them. However, many practical difficulties arise in statically analyzing a binary file.

1.1 Problems with existing formats

Some of the difficulties encountered with current binary formats are:

- *Distinguishing code from data:* The fundamental problem in decoding machine instructions is that of distinguishing code (i.e, instructions) from data within an executable. Machine code in the text segment often contains data embedded between machine instructions. For example, in the C programming language, typical compilers generate code for `case` statement as an indirect jump to an address loaded from some location in a *jump table*. This table, which contains the target addresses, is also placed along the instructions in the text segment. Another example is that of the data inserted in instructions for alignment purpose, presumably to improve instruction-fetch hit rates. Such data causes disassembly problems in architectures such as Intel x86, which has dense instruction set. Thus, most data bytes are likely to appear as valid beginning bytes of instructions. This is a major source of problem for disassembly based on *linear sweep* algorithm [37], which in the process of decoding bytes sequentially misinterprets the embedded data as instructions.

- *Indirect jumps/calls:* One of the ways to avoid misinterpretation of data as instructions is to use *recursive traversal* disassembly algorithm [37] in which disassembly starts from the entry point of the program, and whenever there is a jump instruction, it continues along the control-flow successors of the instruction. However, this approach fails to obtain complete disassembly in presence of indirect jumps because of the difficulty involved in identifying the targets of the instructions. A similar difficulty to statically predicting the targets of function calls is presented by indirect call instructions.

- *Variable-length instruction sets:* Unlike RISC architectures, in which all instructions are fixed-sized, CISC architectures (such as x86) often have variable-length instructions, which complicates their disassembly. In presence of variable-length instructions, a single disassembly error increases the likelihood of errors in disassembly of many of the subsequent instructions. On the other hand, a disassembly error in fixed-length instructions does not propagate to subsequent instructions.

- *Distinguishing address and non-address constants:* It is difficult to distinguish between addresses and non-address constants. Making the distinction is necessary in order to perform any modification to a binary which causes code or data to be relocated. For existing binary formats, there is no general mechanism to correctly make this distinction in every case.

- *Instructions generated through non-standard mechanisms:* Sometimes executables contain instructions generated through non-standard mechanisms (such as hand-written assembly

code). Such instruction sequences may violate high-level invariants that one normally assumes hold true for compiler generated code. For example, in many mathematical libraries it not uncommon for control to branch from one function into middle of another, or fall through from one function into another, instead of using a function call. This kind of code complicates analysis of binaries considerably.

1.2 Security applications

The ability to perform analyses on binaries is very useful from a security viewpoint. Some of the applications for which binary analyses could be useful are:

- **Intrusion Detection**. The ability to derive program behavior models from binaries is useful for detection of anomalous behavior of programs. Such models have been generated from the program source [43], or from runtime traces [17, 39]. However, the application of binary analysis techniques in intrusion detection has been limited due to the difficulties cited above.

- **Retrofitting binaries for memory safety**. To prevent memory related errors in programs written in type-unsafe languages like C/C++, source transformation techniques have been proposed [31]. Similar techniques for rewriting binaries are desirable.

- **Static verification of binaries**. Binaries could be statically checked for temporal safety properties. In fact, proofs of such temporal safety properties (and memory and type safety properties) could be generated if a sound analyses of binaries is possible.

From this discussion, it is clear that the analysis of binaries is needed for various security analyses, however, in their current form it is not practical. Hence, there is a clear need for a mechanism that can address the above-mentioned problems and enable a sound analysis of binaries. However, proposing a standard that is radically different from existing standards will only make the job of transitioning to the new standard difficult, and unlikely to have a broad impact.

Several attempts have been made to perform analysis and transformation of binaries (see Section 2 for more details). Unfortunately, due to the difficulties involved, the resulting analyses are incomplete and do not provide the strong guarantees required for security.

To summarize, the ideal standard for a security-enhanced binary framework must possess the following properties:

- *Suitability for analyses.* It should clearly address all the problems listed above, thereby making the binary suitable for sound analysis, such as generation of models of security relevant behavior of the code for static checking.

- *Compatibility with existing formats.* It should be seamlessly inter-operable with existing binary standards. This way the format will allow an easy and gradual migration from existing standards.

- *Less stress on existing compiler infrastructure.* The new format should introduce very little work on the part of compiler

infrastructure so that it is easily adaptable by existing compiler developers.

The standard we propose in this paper is a first-step towards achieving these objectives.

2. RELATED WORK

There has also been a significant amount of work done in the area of tools to support *binary editing*. Of these, *QPT* [25], *alto* [29] and *OM* [42] and *EEL* [24] target RISC architectures. *PLTO* [12], and *LEEL* [46] target x86 ELF binaries. For the Windows environment, *Etch* [36] is a tool that targets x86 binaries and *Vulcan* [13] works with x86, IA64 and MSIL binaries. *Dyninst* [4] supports analysis and instrumentation of code at runtime. *UQBT* [8] is an architecture-independent binary translation framework. Unfortunately, none of these tools can distinguish code from data in all cases, so they are not guaranteed to work for every binary. This latter restriction applies to all the existing work, in particular for the x86 architecture, it is not possible to distinguish code from data unless the compiler obeys certain code generation restrictions, or provides some auxiliary information.

Typed Assembly Language (TAL) [27] adds type annotations and typing rules to assembly language. While TAL is an extensive body of work that produces verifiably safe code, it is an entirely different target platform, that requires whole scale rewriting of existing compiler infrastructure. On the other hand, we target existing systems and target platforms to trade-off between practicality and strong verifiable guarantees.

There are also a number of other approaches to preventing low level memory related programming error exploits. *Static analysis* has been used to detect memory errors at compile-time. Work in this area includes *Splint* [23, 15], *CQual* [41] and *BOON* [44]. Most of these techniques are limited by the availability of source code for the programs that are analyzed.

In addition, static analysis and verification have been used to prove safety properties of programs. Proof carrying code [30], MOPS [6], and metacompilation [20] are all examples of techniques that ensure program properties through static analysis and verification. Most of these approaches depend on availability of source code or using a type safe language. Model carrying code [40] could be used to verify program properties by generating models from binaries, but the accuracy of such models depends very much on the auxiliary information that is provided along with SELF.

Runtime checking uses inserted checks to detect memory errors before they can be exploited. Work in this area for low level memory safety includes *bcc* [9], *Purify* [21], *Safe-C* [2], *CCured* [31] and *runtime type checking* [26]. Also in this category is the *CodeCenter* interpretive debugger [22]. These approaches all introduce high runtime overheads of at least 100%, making them useful for debugging and testing, but not for incorporating into production binary releases.

In addition, runtime monitoring for safety has been used to ensure access control, resource access and temporal safety properties. Java [19], Naccio [16], and SASI [14] are all examples of systems that perform runtime checks. SASI operates on binaries and performs code transformation. However, as mentioned earlier, its success is largely dependent on the presence of invariants obeyed by

the code generation strategy of the `gcc` compiler. The modifications we suggest to ELF would be beneficial to SASI and similar techniques, as they could leverage the information in the extra section while performing the analysis for the transformation.

3. SELF – AN ENHANCEMENT TO ELF

In this section, we describe the enhancement to the ELF binary file format which will enable the analysis and transformation of binary code. The extension is simple; yet, it enables a wide range of sound program analyses to be performed simply by addressing the drawbacks that were presented in Section 1.

Our discussion is centered around the ELF file format and we shall exclusively describe it in the context of the x86 architecture. However, the general principles behind this discussion are applicable to other formats and architectures.

Before describing our extension, we shall briefly describe the ELF format. (See [32] for a detailed discussion.)

3.1 ELF format

ELF files fall into the following three types:

- *Executable files* containing code and data suitable for execution. This specifies the memory layout of the process image of program.

- *Relocatable (object) files* containing code and data suitable for linking with other object files to create an executable or a shared object file.

- *Shared object files* (shared library) containing code and data suitable for the link editor (`ld`) at the link-time and the *dynamic linker* (`ld.so`) at runtime.

A binary file typically contains various headers that describe the organization of the file, and a number of sections which hold various information about the program such as instructions, data, read-only-data, symbol table, relocation tables and so on.

Executable and shared object files (as shown in the execution view of Figure 1) are used to build a process image during execution. These files must have a *program header table*, which is an array of structures, each describing a segment or other information needed to prepare the program for execution. An *object file segment* contains one or more sections. Typically, a program has two segments: (1) a code segment comprised of sections such as `.text` (instructions) and `.rodata` (read-only data) (2) a data segment holding sections such as `.data` (initialized data) and `.bss` (uninitialized data). The code segment is mapped into virtual memory as a read-only and executable segment so that multiple processes can use the code. The data segment has both read and write permission and is mapped exclusively for each process into the address space of that process.

A relocatable file (as shown in the linking view of Figure 1) does not need a program header table as the file is not used for program execution. A relocatable file has sufficient relocation information in order to link with other similar relocatable files. Also, every relocatable file must have a *section header table* containing information about the various sections in the file.
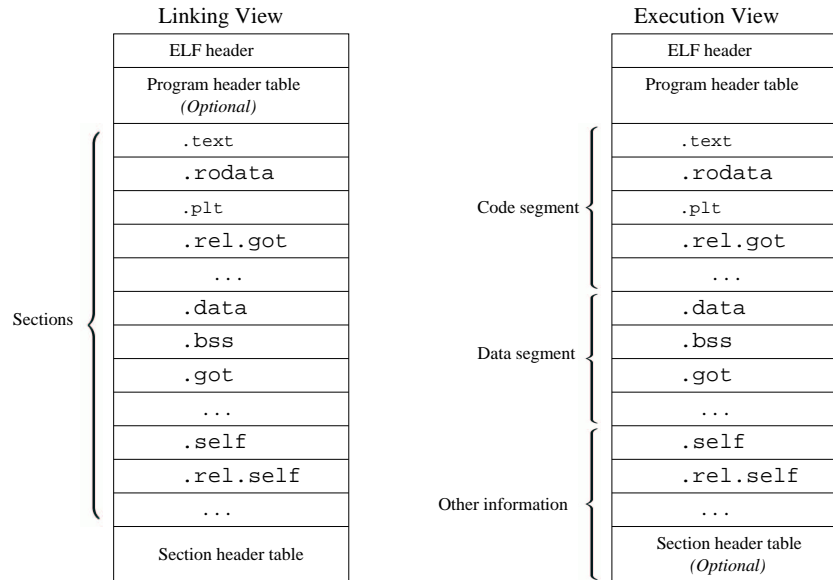
	Linking View			Execution View	

ELF header
Program header table *(Optional)*
.text
.rodata
.plt
.rel.got
...
.data
.bss
.got
...
.self
.rel.self
...
Section header table

ELF header
Program header table
.text
.rodata
.plt
.rel.got
...
.data
.bss
.got
...
.self
.rel.self
...
Section header table *(Optional)*

Figure 1: Format of a typical SELF object file.

By default, an ELF executable file or a shared object file does not contain relocation information because it is not needed by the loader to map the program into process memory. Relocation information identifies address dependent byte-streams in the binary that need modification (relocation) when the linker re-maps the binary to different addresses. A single entry in a relocation table usually contains the following: (1) an offset corresponding to either the byte-offset from the beginning of the section in a relocatable file, or the virtual address of the storage unit in an executable file, and (2) information about the type of relocation (which is processor specific) and symbol table index with respect to which the relocation will be applied. For example, a call instruction's relocation entry would hold the symbol table index of the function being called.

Many binary tools rely on relocation information for analysis and transformation of binaries. Transformation of a binary file often requires modifications in which the subsequences of the machine code are moved around. When this is done, the data referenced by relocation entries must be updated to reflect the new position of corresponding code in the executable. In the absence of relocation information, binary tools resort to nontrivial program analysis techniques [24, 8, 46, 34]. These techniques are inadequate and hence the tools adopt conservative strategies, thereby restricting the their efficacy in performing various transformations. Also, due the fact that relocation information is not required for execution, many linkers do not have option flags to retain the information in the executables. In addition, even the presence of relocation information does not help in certain kinds of binary transformations. In particular, the relocation table does not give sufficient information about the data and instructions used in the machine code. Hence, certain transformations which require complete disassembly of instructions and modification of data are not possible. The SELF extension, described in the following section, is specifically intended to deal with this problem.

3.2 SELF extension
The SELF extension will reside in a section named `.self` which will be indicated by the section head table in both execution and linking views as shown the Figure 1. The purpose of the extension is to provide additional information about instructions and data used in various sections. As discussed before, much of the infor-

mation is available in relocation tables and symbol table. An object file compiled with debug flag option, contains debug information which can also provide useful information such as type and size of data, addresses of functions, etc. However, in a typical software distribution model, binary files are compiled with optimization which renders debug information incorrect. Also, binaries are stripped, which means that they do not have a symbol table. Here, the objective is to distribute slim and efficient binaries containing no superfluous information and which are not easy to reverse engineer. The SELF extension is designed with these objectives in mind. It concisely captures only the relevant information required to perform post-link-time transformations of binary code. This information is described in the form of a table of *memory block descriptors*. A memory block is a contiguous sequence of bytes within a program's memory. Each memory block descriptor has four fields, as shown in Figure 2. The fields are interpreted as follows:

1. *Memory Tag* - type of the block of memory. This includes various kinds of data and code and their pointers. Also includes a bit which indicated whether or not it is safe to relocate the block.

2. *Address* - the virtual address of a block of memory within the data or code of the shared object/executable. This field is meaningful only for executable or shared object files where locations of code and data of the program have been finalized.

3. *Alignment* - this indicates alignment constraints for certain type of data or instructions.

4. *Width* - size of data for the entries that correspond to certain data related memory tag.

During code generation, the compiler adds entries to the table depending upon the *memory tag* of data or instructions. The memory tags fall into the following categories:

- *Data stored between instructions*. This memory tag corresponds to data used in the code, either in the form of jump

Tag	Address	Alignment	Width

Field	Meaning
Tag	Summary of block contents
Address	Starting address of block
Alignment	Alignment requirements of block
Width	Block size in bytes

Figure 2: Layout and interpretation of a SELF memory block descriptor.

tables or padding bytes which are used to enforce alignment restrictions. This helps to identify and disassemble all of the machine instructions in the program.

- *Code address*. Code addresses appear in the program mainly in the form of operands corresponding to targets of jump or call instructions. The addresses could be used in instructions either as relative displacements or as absolute values stored in register or memory. Also, there could be other types of instructions which use code addresses. Typical examples are (1) a PUSH instruction used to pass a constant function-address as a parameter to a callback function and (2) code addresses contained in jump tables. During transformation of binaries, the code at these addresses might be relocated. Therefore, such operands or locations must be changed to point to the new address.

- *Data address constant*. Static data in the program is referenced using data address constants in the instructions. Entries of these types are required if the data segment of the binary undergoes reorganization.

- *Offset from GOT (global offset table)*. This corresponds to the constant offsets which are used to access static data in position independent code (PIC). Such offsets will be modified if the GOT or the data is relocated.

- *Offset used to obtain base address of GOT*. This pertains mainly to x86-specific position independent code generated for shared objects. For this purpose, the code is generated in such a way that the %EBX register is assigned the value of the GOT's base address. During the generation of this code, a constant value is used which corresponds to the relative offset of the program counter from the GOT. This constant requires modification if the GOT or the code containing the program counter undergoes relocation during binary transformation.

- *PLT (procedure linkage table) entry address*. In an executable or a shared library, a position-independent function call (e.g., a shared library function) is directed to a PLT entry. The PLT entry in turn redirects it to its absolute location, which is resolved by dynamic linker at run time. Code addresses associated with these function calls need different memory tags as some binary transformation may require relocation of only the PLT.

- *Offset from frame pointer*. This memory tag identifies the locations of constant offsets from the frame pointer (%EBP) that are used by instructions which access stack-allocated objects. These constants have to be changed if there is a binary transformation that relocates stack-allocated objects.

- *Routine entry point*. This memory tag identifies the entry points of all the routines in the code segment.

- *Stack data*. Stack data is mainly associated with the local variables of functions in the program. Memory for such data is allocated on the stack dynamically during function invocations. Therefore, the virtual memory addresses of stack data can not be determined statically. However, each stack datum is allocated on the stack at a fixed constant negative offset from the frame pointer. The *address* field in an entry of this tag contains this offset instead of the virtual memory address.

- *Static data*. Static data corresponds to different storage units allocated in the code segment for global or static variables used by the program. This memory tag is used to identify the locations of each such storage unit in the code segment.

Apart from these, there are other memory locations which contain data required for dynamic linking. The entries of these types are retained in the binary file and hence we do not require to save them separately. The above types are all that is needed to effectively disassemble the executable and thereby make program analysis and transformation of the executable possible.

For static or stack-allocated data, additional information is available through the fields *alignment* and *width* of the entries. A compiler generates the memory layout of program data depending on their types. Data could either have scalar or aggregate types. A datum of a scalar type holds a single value, such as an integer, a character, a float value, etc. An aggregate type, such as an array, structure, or union, consists of one or more scalar data type objects. The ABI of a processor architecture specifies different alignment constraints for different data types in order to access the data in an optimum way. Scalar types align according to their natural architectural alignment, e.g., in the Intel IA-32 architecture, the integer type requires word (4 byte) alignment. The alignment of an aggregate type depends on how much space it occupies and how efficiently the processor can access the individual scalar members objects within it. The data entries hold alignment and width of only the scalar and aggregate objects and not for the members inside the aggregate objects. Thus, relocation can be performed only on the scalar or on the aggregate objects as a whole.

4. SELF GENERATION

Figure 3 shows the distribution model for SELF binaries. Notice that there are two paths in the distribution mechanism. The shorter path represents the case where a modified compiler is used to generate SELF binaries at compile-time, directly from the source code. The longer path refers to the binaries generated through a standard (i.e., not modified) compiler, as occurs when a provider is unable or unwilling to use an augmented compiler, and the source code
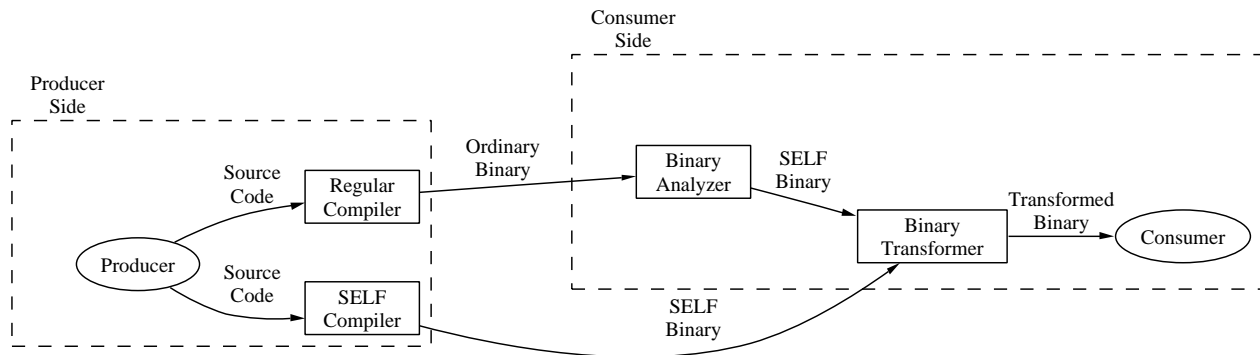
Figure 3: Model for the distribution of SELF binaries.

is not available at the consumer end. In this case, binary analysis techniques are used to generate the SELF section. There are three primary reasons for the dual-path approach:

- It allows externally supplied and pre-existing legacy binaries to be supported to as great a degree as possible.

- It allows for complete support of SELF-compiled binaries.

- The SELF-analysis and generation is decoupled from the binary transformation, resulting in better performance in cases where programs are transformed multiple times (e.g., as is the case with address obfuscation, discussed in Section 5).

The rest of this section describes these two paths in greater detail.

4.1   Compile-time SELF generation

Generating SELF from within a compiler is a straightforward process, as most of the information required can be gleaned directly from the compiler's internal symbol tables. Also required will be a `.rel.self` section, which will contain the relocation entries used by the linker to update the `.self` section when the program layout is finalized. A good implementation strategy for adding a SELF generation option to a typical compiler is to modify the code used to generate debugging information, since there is much overlap between the debugging information and SELF. The `.self` section contents can be viewed as a copy of the debugging information with unneeded information removed, such as variable names and types, and extra information added, such as information about pointers embedded within machine instructions.

4.2   Post-compilation SELF generation

Post-compilation SELF generation poses a much greater challenge, since the binary file must be analyzed. In particular, the analysis must identify the following:

- Data embedded within the code segment

- Pointer values (within both the data and code segments)

- The size of each data object

4.2.1   Identifying data within the code segment

Data embedded within the code segment is identified by starting with a set of known instructions (i.e., the set of known entry points), analyzing each known instruction to find its set of successors, and

repeating this process until the transitive closure is reached. As mentioned earlier, on the x86 architecture this analysis is actually somewhat difficult, primarily due to indirect calls and jumps, the use of runtime-generated code on the stack, and variable-length instructions.

One tool which already does a fairly good job of distinguishing code from data is LEEL [46], although there is still some room for improvement. In particular, in cases where some branches and/or entry points are uncertain, the set of feasible disassemblies can be computed (i.e., those disassemblies which don't branch out of the address space or collide with a known data value). If more than one disassembly is feasible, then each byte can be marked based on how it is utilized in each disassembly, according to the following rules:

- A byte is *definitely code* if it can be executed under every feasible disassembly.

- A byte is *definitely data* if it cannot be executed under every feasible disassembly.

- A byte is *indeterminate* (either data or code) if exactly one of the above two conditions does not hold.

This approach yields the most information possible without making any assumptions about the behavior of the code generator, given the difficulties inherent in disassembling x86 machine code.

4.2.2   Identifying pointer values

Identifying pointer values within the code and data segments is done by flow analysis. Instructions which dereference pointer values are traced backwards to discover the origin of the pointer value. This process can essentially be viewed as a type inference problem. Values which are loaded from code or text segment and then used as pointers along every execution path can be inferred to definitely be pointers; and similarly values which are used solely as data along every execution path can be marked as definitely data. Values which propagate only through static memory and registers are easy to correctly type using this approach; values which propagate through the stack are slightly harder; and values which propagate through the heap are rather difficult. The end result of the analysis is that every data value is typed as being either a definite pointer, definite scalar, or indeterminate/both.

4.2.3 Identifying data size

Upper bounds on the size of each data object are computed by analyzing the instructions which access them. The challenge is to distinguish structure fields from atomic variables. This can be done by first analyzing every pointer argument to all routines in a call-path-insensitive manner to determine the range of offsets that are accessed via the pointer. The second step is to use the function argument information in an analysis to determine the potential range of each pointer.

Values stored over the access range of a pointer are likely to be part of the same array or structure (since it's highly unlikely that any linked data structures would be stored in static memory) and should not be relocated except as a block. On the other hand, even if a value is an array or structure, as long as its components are accessed individually and not via pointer arithmetic, then it is okay to split up the aggregate into separate components. The problem lies with pointer dereferences whose range can't be determined. To help with this, for each pointer dereference operation, the range of the values that could be accessed is expressed as one or more of the following:

- $Heap$ – the pointer points somewhere on the heap.

- $Stack : \langle low \dots high \rangle$ – the pointer points to some range of local (stack) storage.

- $Static : \langle low \dots high \rangle$ – the pointer points to some range of static (data or code) storage.

This approach allows pointers which point only to stack and/or heap locations to be identified and safely ignored. Greater precision can be achieved by making the analysis call-path sensitive, and/or incorporate additional constraint analysis, which would allow for discovering properties such as the fact that bzero(x, n) overwrites values from x to x + n - 1. A reasonable compromise is to encode such constraints for known libc calls such as bzero.

5. EXAMPLE: ADDRESS OBFUSCATION

Memory-error exploits, such as buffer overflow, and format-string attacks, are the one of the most common classes of attack on the Internet today. The prevalence of these attacks is due to several factors. First, memory errors are commonplace, due to the prevalence of non-memory safe languages (primarily C and C++). These languages are popular to due the fine-grained control they give the programmer over a system's memory, but unfortunately, they also leave the burden of performing safety checks on pointer and array accesses up to the programmer. Second, memory errors such as unchecked array accesses often result in security vulnerabilities which enable an attack to be remotely launched from across a network. For example, it is common for a fixed size, stack-allocated buffer to be used to hold data transmitted from the network. If the programmer forgets to include runtime checks to ensure that incoming data is not larger than the array, then the incoming data may overflow the buffer, going past the end of the array and eventually reaching the stored return address of the current function. This is the mechanism by which the infamous *buffer overflow* attack works [33, 28].

An observation that one can make about the buffer overflow attack is that it is *absolute address-dependent*: the attacker must know the absolute address at which the injected code will reside (typically the starting address of the buffer), and then overwrite the return address with the injected code address.

An additional observation that is important to keep in mind is that buffer overflows are not the only possible memory-error exploits. In fact, there are many other possible attacks, which can target any region of a program's memory, and may in some cases only depend on the relative distances between two items. Some examples of these include:

- Overflowing from a buffer onto a string which will be passed to an execve system call. This only depends on knowing the distance between the string and the buffer, and is hence *relative address-dependent*. Such attacks could potentially occur on the stack, on the heap, or in static storage.

- Using a *format-string* attack to replace the address stored in a function pointer [38, 35]. The attack is absolute address-dependent, since it requires knowing the absolute address of the function pointer and the called code. The function pointer itself could be stored in any of the program's data regions, and the code could be library code, program code, or injected code.

- Due to the lack of adequate checking done by malloc on the validity of blocks being freed, code which frees the same block twice corrupts the list of free blocks maintained by malloc. In the case where the doubly-freed block contains an input buffer, this corruption can be exploited to overwrite an arbitrary word of memory with an arbitrary value [1]. This attack is absolute-address dependent.

The point of these examples is to illustrate that the classic buffer overflow attack is just one of many possible attacks, and that solutions which only protect against a limited number of exploits, such as overflowing onto the return address [11, 10] are only partial solutions which will simply force attackers to be more resourceful in their search for other memory errors to exploit [5]. Instead what is needed is an approach which protects against the full spectrum of potential memory-error exploits.

One way of achieving something close to full-spectrum protection from memory error exploits is to make it impossible for an attacker to reliably know any absolute or relative address within a program, thereby thwarting attacks which are absolute or relative address-dependent. That is the essential idea behind *address obfuscation* [3], a technique in which the memory locations of the data and code of a program are randomly relocated prior-to and during each execution. The only available option address obfuscation leaves attackers with is to make random guesses. Furthermore, it has been shown in [3] that address obfuscation can be implemented in a fairly lightweight manner (requiring no compiler or kernel modifications), while requiring an attacker to make of the order of 10^4 attempts before success is likely to occur, with failed attempts resulting in conspicuous program crashes, making intrusion detection fairly easy.

The largest obstacle towards the wide adoption of address obfuscation is the difficulty of applying it to shrink-wrapped binaries. Work to date has either required source code access [18], or has been restricted from performing the full subset of possible obfuscations due to the intractability of analyzing binary code [3, 45,

7]. However, with SELF binaries, complete address obfuscation of binaries is feasible, providing the following benefits:

- All attacks which exploit memory errors will become non-deterministic, with a small chance of success (dependent on availability of virtual address space, but 1 in 10^4 can be easily achieved [3]). Failed attacks will typically result in system crashes which, will make the attack attempts easily detectable.

- In addition to buffer overflows, attacks which target other regions of memory will be prevented, including many attacks which haven't been discovered yet, but are likely to become popular as the techniques which prevent the return address from being overwritten [11] become widely adopted.

- For additional security, the obfuscation can be combined with other runtime security techniques, such as stackguard [11].

- The runtime overhead will be low, requiring an initial startup cost, but very little cost after that. In cases where the startup cost is unacceptable, the obfuscation can be done once statically on the binary file, and the file can be re-obfuscated on a regular basis to deter attacks.

- Minimal changes to existing compilers are required. All that is needed is that the compiler create the extra .self section containing information similar to what is already provided to the linker; no changes in code generation are required.

- The augmented SELF binaries will execute transparently on systems which don't support obfuscation (since they are backwards-compatible with ELF binaries).

- The degree of obfuscation can be tailored according to the desires of the user, as can the time when the obfuscation occurs (load time or statically prior to execution). Statically obfuscated binaries can be re-obfuscated as often as desired.

- The obfuscation can be done by a special loader which runs on top of the kernel, so no kernel modifications are required; however, integration with the kernel is still possibility for those who desire it.

The obfuscation is achieved as follows. First, the organization that produces and/or distributes a program (henceforth the *code producer*) uses an augmented compiler to generate a SELF binary. The SELF binary is now ready for widespread distribution.

Next, the host/person using the program (henceforth referred to as the *code consumer*), downloads the SELF binary and runs it through the *obfuscation transformer*. The transformer reads the .self section and randomly relocates the program's data while inserting code to perform additional relocations as the program executes. The consumer may run the software through the transformer as often as desired to deter persistent attackers (this might be setup as a regular job via the cron daemon).

5.1   The obfuscation transformer
The transformer first reads the .self section. It then performs the following transformations:

- *Stack base address randomization.* This transformation is done by inserting code before main is called which subtracts a large random value (to insert a large gap) from the stack pointer at runtime. This consumes virtual address space, but not much actual memory. Additionally, this gap is made un-writable in order to prevent very large buffer overflows (see [3] for details as to why such overflows are a concern).

- *Stack relative address randomization.* This transformation is done by increasing the size of each stack frame (by simply changing the constant added to the stack pointer in the function preamble), then randomly moving all variables within the frame. Furthermore, arrays are located to addresses higher than non-arrays, to reduce the number of potential targets of an overflow. All instructions which fetch and store values from/to the stack are patched to reflect the new offsets of each variable within the frame.

- *Code base address randomization.* This transformation is done by changing the virtual address at which the code is loaded, then relocating all absolute addresses within the code to reflect the new base address.

- *Data address randomization.* This transformation is done by changing the address at which the the data segment is loaded, then randomly reordering the variables, and finally introducing a random amount of padding between variables. All instructions which access a static datum are patched to use the new address of that datum.

- *Heap address randomization.* This transformation is achieved as a side-effect of the data segment base address randomization, since the heap's starting address follows the end of data segment on Linux systems. Additionally, code is inserted to increase heap allocation requests by a small random amount, thereby randomizing relative addresses within the heap.

The effect of these transformations is to make the absolute address of all code and data objects unpredictable. Furthermore, the relative distance between any two data items is unpredictable as well, regardless of which region the data is stored in. As explained earlier in this section, the unpredictability provides protection from absolute- or relative-address dependent attacks, such as buffer overflows.

5.2   Other techniques
In addition to address obfuscation, a number of other analyses and transformations are possible with SELF binaries. These include the ability to extract control-flow graphs, call graphs and other models of program behavior; the ability to correctly insert inline reference monitors, such as monitors which use a finite-state automaton to enforce restrictions on temporal orderings of system calls; the ability to perform post-compilation optimizations; and translation into other languages/architectures, such as converting x86 code into SPARC code. All of these techniques require the ability to distinguish a program's code from its data, and hence cannot be done soundly on ordinary binary files, but with SELF binaries they are tractable.

6.   CONCLUSION
As we have shown, SELF is a relatively simple extension to the existing ELF binary distribution format which enables a number of security techniques to be applied in the absence of source code.

Previous work on program security has almost exclusively focused on the analysis and transformation of program source code; this has been due to the impossibility of soundly analyzing existing binary file formats. Rather than accepting this limitation, SELF provides the information required to apply many of the existing techniques to binaries, thereby allowing them to have a much broader impact. Our hope is that this approach will allow advanced language-based security techniques to be applied in the real-world user environment, where source code is unavailable.

7. REFERENCES

[1] Anonymous. Once upon a free . . . . *Phrack*, 11(57), August 2001.

[2] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 290–301, Orlando, Florida, 20–24 June 1994. *SIGPLAN Notices* 29(6), June 1994.

[3] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceedings of the USENIX Security Symposium*. USENIX, 2003 (to appear).

[4] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[5] Bulba and Ki13r. Bypassing stackguard and stackshield. *Phrack*, 11(56), 2000.

[6] Hao Chen and David Wagner. Mops: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 235–244. ACM Press, 2002.

[7] Monica Chew and Dawn Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Carnegie Mellon University, December 2002.

[8] Cristina Cifuentes, Mike van Emmerik, Norman Ramsey, and Brian Lewis. The university of queensland binary translator (uqbt) framework. Technical report, The University of Queensland, Sun Microsystems, Inc, 2001.

[9] Samuel C.Kendall. Bcc: run–time checking for c programs. In *Proceedings of the USENIX Summer Conference*, El. Cerrito, California, USA, 1983. USENIX Association.

[10] Tzi cker Chiueh and Fu-Hau Hsu. Rad: A compile-time solution to buffer overflow attacks. In *21st International Conference on Distributed Computing*, page 409, Phoenix, Arizona, April 2001.

[11] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, jan 1998.

[12] Saumya Debray, Benjamin Schwarz, and Gregory Andrews. Plto: A link-time optimizer for the intel ia-32 architecture. In *Proc. 2001 Workshop on Binary Rewriting (WBT-2001)*, Sept. 2001.

[13] Andrew Edwards, Amitabh Srivastava, and Hoi Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, April 2001.

[14] Ulfar Erlingsson and Fred B. Schneider. Sasi enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigm Workshop Ontario, Canada*, 1999.

[15] David Evans. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 44–53, Philadelphia, Pennsylvania, 21–24 May 1996. *SIGPLAN Notices* 31(5), May 1996.

[16] David Evans and Andrew Tywman. Flexible policy directed code safety. In *Proceedings of the 1999 IEEE conference on Security and Privacy*, 1999.

[17] S Forrest, S Hofmeyr, and A Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 1998.

[18] Stephanie Forrest, Anil Somayaji, and David H. Ackley. Building diverse computer systems. In *6th Workshop on Hot Topics in Operating Systems*, pages 67–72, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[19] L Gong, M Mueller, H Prafullchandra, and R Schemers. Going beyond the sandbox: An overview of the new security architecture in the java development kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[20] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI-02)*, volume 37, 5 of *ACM SIGPLAN Notices*, pages 69–82, New York, June 17–19 2002. ACM Press.

[21] Reed Hastings and Bob Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In USENIX Association, editor, *Proceedings of the Winter 1992 USENIX Conference*, pages 125–138, Berkeley, CA, USA, January 1992. USENIX.

[22] Stephen Kaufer, Russell Lopez, and Sesha Pratap. Saber-C — an interpreter-based programming environment for the C language. In USENIX Association, editor, *Summer USENIX Conference Proceedings*, pages 161–171, Berkeley, CA, USA, Summer 1988. USENIX.

[23] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Washington, D.C., August 2001.

[24] J. R. Larus and E. Schnarr. Eel: Machine-independent executable editing. In *In Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, June 1995.

[25] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Software - Practice and Experience*, 24(2):197–218, 1994.

[26] A. Loginov, S. Yong, S. Horwitz, , and T. Reps. Debugging via run-time type checking. In *Proceedings of FASE 2001: Fundamental Approaches to Software Engineering*, April 2001.

[27] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. In *Twenty-Fifth ACM Symposium on Principles of Programming Languages*, San Diego, CA, Jan 1998.

[28] Mudge. How to write buffer overflows. http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html, 1997.

[29] Robert Muth, Saumya K. Debray, Scott A. Watterson, and Koenraad De Bosschere. alto: a link-time optimizer for the compaq alpha. *Software - Practice and Experience*, 31(1):67–101, 2001.

[30] G Necula. Proof-carrying code. In *ACM Principles of Programming Languages*, 1997.

[31] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages (POPL '02)*, pages 128–139, Portland, OR, January 2002.

[32] Mary Low Nohr. *Understanding ELF Object Files and Debugging Tools*. Prentice Hall Computer Books, 1993.

[33] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.

[34] Manish Prasad and Tzi cker Chiueh. A binary rewriting defense against stack-based buffer overflow attacks. In *Usenix Annual Technical Conference (to appear)*, June 2003.

[35] Riq. Advances in format string exploitation. *Phrack*, 11(59), 2002.

[36] Ted Romer, Geoff Voelker Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian N. Bershad, and J. Bradley Chen. Instrumentation and optimization of Win32/Intel executables using etch. pages 1–8.

[37] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. Disassembly of executable code revisited. In *Proc. 2002 IEEE Working Conference on Reverse Engineering*, October 2002.

[38] scut. Exploting format string vulnerabilities. Published on World-Wide Web at URL http://www.team-teso.net/articles/formatstring, March 2001.

[39] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. A fast automaton-based approach for detecting anamolous program behaviors. In *IEEE symposium on security and privacy*, 2001.

[40] R Sekar, V.N. Venkatakrishnan, Samik Basu, Sandeep Bhatkar, and Daniel C. DuVarney. Model carrying code: A practical approach for safe execution of untrusted applications. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[41] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proccedings of the 10th USENIX Security Symposium*, Washington, D.C., August 2001.

[42] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, December 1992.

[43] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE symposium on security and privacy*, 2001.

[44] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, San Diego, CA, 2000.

[45] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. Manuscript submitted for review.

[46] Lu Xun. A linux executable editing library. Masters Thesis, 1999. available at http://www.geocities.com/fasterlu/leel.htm.