Byte Bouncer: Efficient Enforcement of Least Privilege Using Certified Binaries

Arindam Chakrabarti, Bor-Yuh Evan Chang, and Adam Chlipala

University of California, Berkeley, California, USA
{arindam,bec,adamc}@cs.berkeley.edu

CS261: Security in Computer Systems
Fall 2004

Abstract. Almost all computer users today are aware that malicious code, such as viruses and worms, can cause a great amount of damage. Nonetheless, most software is still distributed as binary executables with basically no certification of their safety, nor do users use sufficient safeguards when executing such untrusted code. We assert that while end users know the security properties they would like upheld, most existing systems do not provide a method by which those precise properties can be specified. In this paper, we attempt to address this problem with an instantiation of Proof-Carrying Code for high-level security policies. We give a policy description language that can be used to describe security policies of interest to the end user in terms of logical pre- and post-conditions on system calls, as well as an enforcement mechanism to ensure conformance to the policy.

1 Introduction

With the publicity of Internet viruses, worms, and "spy-ware" in recent years, the majority of computer users have come to understand that while software can provide an extraordinary amount of functionality, bugs or code with malicious intent can cause great harm. Yet the state of practice for distributing and acquiring software is simply to ship binary executables, with very few safeguards applied by end users. Only recently have some commercial operating systems begun to prompt the user when starting a potentially untrusted executable.

Most existing systems, such as most operating systems or even Java stack inspection [WBDF97], tackle this problem using the abstraction of "privileges"; however, privileges are often too coarse-grained to express what users really want to allow. For instance, a root mail daemon may need to execute code as other users to apply filters, but it should not run one user's filter as a different user. Trying to make privileges more specific only leads to an infinite regress; there is always some program that does not fit into the privilege-management framework that users find acceptable due to more subtle properties of its interactions with the system.

The underlying problem here is a common one—conflating policy and mechanism. End users do not care how security is achieved, as long as it is achieved. The end user generally knows the security policy he would like enforced (*e.g.*, "this particular program should only read and write files within in the `/tmp` directory"), while he does not know or care how it is enforced (*e.g.*, the program is run in a state where only the `/tmp` directory is accessible using, say, `chroot`).

In this paper, we propose to apply static analysis to the enforcement of such high-level security policies. We suggest making it the responsibility of the *end user* to specify a security policy, while requiring that the *code producer* show why it is satisfied, in the style of Proof-Carrying Code (PCC) [Nec97,AF00]. Past work in PCC has applied mostly to memory safety and other fixed, low-level policies. We put forth a technique towards applying PCC to higher-level security policies specified by the end user.

The primary contributions of this paper are as follows:

– We have prototyped a system for statically verifying proof-carrying binaries for conformance to security policies specified in the form of logical pre- and post-conditions on system functions; and
– We have identified a way for the end user to precisely specify security policies of interest to him, rather than being tied to specifying in terms of particular fixed "privileges".

In Section 2, we characterize the driving force behind our design by providing the threat model and security goals for such a system. We illustrate informally the problem and allude to a solution in Section 3. In Section 4, we give a more detailed description of our policy language and the implementation of the verification procedure. We detail experience from experiments generating and checking proofs for a few publicly-available programs in Section 5. In Section 6, we describe some additional considerations in order to realize this approach from both the code producer and the code consumer's perspective. We summarize related work in Section 7 and conclude with a high level analysis of our approach in Section 8.

2 Security Analysis

2.1 Threat Model

We make the usual assumptions about mobile code security. A *code consumer* will be running some piece of mobile code. The wholly untrusted *code producer* provides this code to the consumer. The provided "code" may be any sequence of bits whatsoever. The consumer hopes that it is a legal native code program for his architecture. In particular, the method by which the code is generated (typically by using a compiler) is not trusted. The channel through which the code is delivered from the *producer* to the *consumer* is not trusted either.

We assume that security breaches are only possible by exploiting the functionality of a fixed set of *foreign functions*. More precisely, we assume that each native code program runs within a *virtual machine* or *sandbox*. Most instructions available to the program can only manipulate its own local state. However, there is a well-defined way for it to request certain non-local actions (*e.g.*, opening a file or starting another process). For the Java Virtual Machine, this comes through calls to native C functions by the standard class library, which is mediated by permission checks. In UNIX, this comes through a well-known set of system calls. In a correct UNIX implementation, the process abstraction is sufficient that our assumption is realistic.

The basic philosophy is that a process is free to do whatever it feels like, so long as this behavior cannot affect any other process directly. Most previous work in mobile code security has focused on memory safety and has been based around certifying compilers for high-level programming languages. In contrast, we more or less assume that an operating system, virtual machine sandbox, or similar system provides us with sufficient isolation between processes that we do not need to worry about memory safety explicitly. We will still need to do much of the work associated with proving memory safety, because we will need to have some idea of the state of registers and memory locations to analyze foreign function calls and avoid buffer overflow-style attacks, but we can allow any memory accesses that can be proved not to effect any foreign calls. There should be no way to cause unsafe behavior without going through the single channel of foreign function calls.

For the initial prototype, we have focused on dealing with `x86` assembly code produced from C programs by `gcc` on the Linux operating system; however, we do not allow the consumer to assume that every program he receives was built by `gcc`. This would not be much help, anyway, since `gcc` allows inline assembly. The producer is free to provide whatever nefarious data he wants, in an effort to subvert the consumer's security. For this project, we will also expand the set of foreign functions to include a set of trusted library functions not provided by the operating system. A summary of what is trusted and untrusted is given in Table 1.

2.2 Security Goals

We want to provide the code consumer with a way to guarantee that particular programs only make particular kinds of foreign function calls. This goal decomposes into two parts.

First, recall that we have placed the responsibility of specifying the security policy on the end user, the one who ultimately cares that the policy is upheld. As such, the method by which security policies are specified must be carefully considered both for expressivity and usability. An end user with a reasonable amount of programming knowledge ought to be able to construct his own policies without much effort. Indeed, we hope to make it feasible for policies to be created and modified quite often.

The handling of Java applets by popular web browsers provides a kind of anti-example here. There is usually one "applet policy" that specifies what

Trusted	Untrusted
The code consumer.	The good intentions of the code producer.
System calls provided by the operating system.	Binary executables provided by the code producer.
A simple proof checker.	The compiler used by the code producer (*e.g.*, `gcc`).
A formalization of the machine semantics for the target architecture (*e.g.*, `x86`).	The proof generation mechanism used by the code producer.
An interpreter that translates security policies to this formalization	The channel through which the mobile code is delivered.

Table 1. Trusted and untrusted components.

kinds of foreign functions any applet may access. If the user wants to run an applet that requires just one more permission to do its job, he is out of luck. He must give the applet full permissions or give up on running it. The extra permission might be to read a particular file, open a particular network connection, or do some other thing that could be described easily by the end user. In such a case, we would like to give the user the option to extend the standard policy to include the extra capability.

There are many different levels of expressiveness for security policies that we might consider. For the initial prototype, we have focused on *stateless* policies, expressed in terms of logical pre- and post-conditions on foreign functions. It would be natural to allow policies to include something like security automata that model state changes resulting from foreign function calls. We leave this for future work, as it would build upon the work we do here for simpler policies. For even greater usability, one may also imagine building defaults and higher-level user interfaces for specifying common policies.

In summary of this requirement, we want a security policy language that is simple enough for end users to be confident that particular policies accurately represent their intentions. We want it to be tractable for semi-expert users to create their own policies without any special training. A final goal is to facilitate expressing as many different kinds of policies as possible, but for the initial prototype, we focus on a particular subset of policies.

Second, we need a framework for checking that a program satisfies a security policy. Here we build upon past work in Proof-Carrying Code (PCC). Our implementation works within the Open Verifier framework [CCNS05] for Foundational PCC. In particular, we ask that the end user trust the following components:

– A *proof checker* for a logic expressive enough to explain why programs satisfy the safety policy;

– A formalization of how programs execute on the target architecture (in our prototype, the `x86`); and
– An interpreter that translates security policies into this formalization

This is the standard kind of trusted base in Foundational PCC [AF00]. It should be reasonably small, so that we can have confidence in its correctness after manual auditing. In particular, we are not asking the end user to trust that a complicated optimizing compiler works correctly. A bug in `gcc` should not provide a way to circumvent a security policy. We also do not assume anything about the soundness of particular *mechanisms* for achieving the policy, since formalizing this introduces more potentially complicated code to audit.

Instead, it is the *code producer*'s responsibility to provide the mechanism for proving policy conformance. The trusted parts of the Open Verifier define a precise logical notion of when a program conforms to a policy, in effect generating a theorem statement per program-policy pair. The code producer should determine what policies his customers are interested in, through some out-of-band mechanism. He should then include with his program a proof that it satisfies all of these policies.

The standard security benefits of PCC also apply here. If a program in fact violates a policy, it will be impossible for the producer to construct a proof to the contrary, so we need not worry about ever running an unsafe program. It may be hard to prove that some programs satisfy some policies, but we assume that the honest code producer has a good idea of why his program is safe. If he does, then it is reasonable to expect that he can prove it.

The big security property that we require of our system is that it be sound. As described above, it should never declare an unsafe program to be safe, no matter what purported proof is attached to it. Completeness is a secondary goal, which we will only be able to hint at through experiments showing that it is tractable to prove the safety of particular programs.

3 A Motivating Example

Suppose the code consumer would like to download and use a program that provides some helpful statistics about the contents of some files on his hard drive. It will write its results to the file `/tmp/results`. He wants to let it read any files it chooses, but it should not be writing to any file except `/tmp/results`. It also should not try any funny business like uploading the results of its analysis to a server somewhere. The following Byte Bouncer policy enforces these requirements:

```
Function open(name, mode)
  Pre StrEq(name, "/tmp/results") || mode == O_RDONLY

Function read(fd, buf, count)
Function write(fd, buf, count)
Function close(fd)
```

```
if (strcmp(name,"/tmp/results") == 0)
  fd = open(name, O_WRITE);
else
  fd = open(name, O_RDONLY);
```

(a) C source

```
  movl   $.LC0, 4(%esp)
  movl   -8(%ebp), %eax  #  name
  movl   %eax, (%esp)
  call   strcmp
  testl  %eax, %eax
  jne    .L9
  # true-case (write)
  movl   $1, 4(%esp)
  movl   -8(%ebp), %eax  #  name
  movl   %eax, (%esp)
  call   open
  movl   %eax, -4(%ebp)  #  fd
  jmp    .L10
.L9:
  # false-case (read)
  movl   $0, 4(%esp)
  movl   -8(%ebp), %eax  #  name
  movl   %eax, (%esp)
  call   open
  movl   %eax, -4(%ebp)  #  fd
.L10:
```

(b) x86 assembly

Fig. 1. Example untrusted program.

The policy declares that foreign functions open, read, write, and close may be called. Furthermore, the open system call should only be allowed when opening /tmp/results or opening in read-only mode. The other functions may be called with any arguments. Note that we do not allow a program to make a call to a foreign function that does not appear in the current policy, so the above policy is enough to enforce that, for instance, the program does not open any network connections.

Suppose the source of the downloaded program contains the fragment of C code in Figure 1 with the corresponding x86 assembly produced by gcc. Given this assembly and an appropriate proof, Byte Bouncer should be able to conclude that the security policy is never violated. To make this possible, we will extend the security policy to mention strcmp as an allowable foreign function, since it will be provided by the standard C library and not the mobile code. We also want to specify enough about its semantics:

```
Function strcmp(s1, s2)
  Post (result == 0) ==> StrEq(s1, s2)
```

Here, we use a post-condition, which only serves to make the code producer's job easier, hopefully by allowing him to assume some extra true statements. We use the built-in predicate StrEq, which expresses equality of strings with respect to the current memory.

4 Design and Implementation

Armed with the intuition of our overall task, we proceed to a more formal description of the Byte Bouncer policy language and a description of the verification procedure.

4.1 Policy Description Language

Following the principle of least-privilege, the security policy specifies the allowable system calls, along with the conditions that must provably hold for them to be called. For familiarity, the conditions on calls are specified using a C-like syntax of boolean expressions. A complete description of the policy language is as follows:

policy	$P ::= \vec{F}$	
function policy	$F ::=$ `Function` $f(\vec{x})\ \vec{c}$	
contracts	$c ::=$ `Pre` b	pre-condition
	$\mid$ `Post` b	post-condition
boolean expressions	$b ::=$ `true` $\mid$ `false`	boolean constants
	$\mid\ !b \mid b_0\ \&\&\ b_1$	boolean connectives
	$\mid b_0 \,\Vert\, b_1 \mid b_0 \Longrightarrow b_1$	
	$\mid e_0 == e_1 \mid e_0\ != e_1 \mid e_0 < e_1$	integer comparisons
	$\mid e_0 <= e_1 \mid e_0 > e_1 \mid e_0 >= e_1$	
	$\mid$ `StrEq`(e_0, e_1) $\mid$ `Prefix`(e_0, e_1)	string comparisons
expressions	$e ::= n$	integers
	$\mid x$	parameters
	$\mid$ `result`	return value
	$\mid "s"$	string constants
	$\mid e_0 + e_1 \mid e_0 - e_1 \mid e_0 * e_1$	arithmetic

We have provided some initial predicate constructors for specifying arithmetic and string properties in our prototype system, though we expect that in a realistic system, more predicate constructors would be needed and perhaps a mechanism for giving user-defined predicates (and their meanings). The choice of predicates to provide for strings has been driven by what may be necessary (or useful) when specifying conditions on files. Predicate `StrEq`(e_0, e_1) checks that e_0 and e_1 are the same string; `Prefix`(e_0, e_1) asserts that a prefix of e_0 is equal to e_1. The special identifier `result` is used to name the return value of the function call used in post-conditions.

4.2 Verification Procedure

As mentioned earlier, Byte Bouncer is an implementation of these ideas within the Open Verifier Foundational PCC framework. In this paper, we will only sketch the high level operation of the Open Verifier and how this work fits in with it. More detail can be found in [CCNS05].

Fundamentally, the Open Verifier is an engine for executing abstract interpretations of assembly programs. The concrete interpretation of a program is dictated by the semantics of the processor it runs on. This semantics involves states that include very specific information, including the precise value of every register and memory slot. Since this state space is too large to explore exhaustively, the standard trick of program verification is to introduce a notion of abstract states, where one abstract state stands for many concrete states. If the set of possible abstract states is small enough, then we can use exhaustive verification to consider all possible executions of a program. If none of these violates the security policy, then we know that no real execution could violate the policy, as long as we chose a sound abstraction. Any use of the Open Verifier involves specifying an abstraction that is useful for proving some property of programs.

While we have described Byte Bouncer in terms of the traditional PCC model of "programs with proofs attached," the Open Verifier actually uses a more efficient interaction model. A piece of mobile code comes with an untrusted program, called an *extension*, that can be run inside a sandbox to construct, in effect, the proof on the consumer's side. The extension's main job is to come up with a system of abstract states and prove that it is sound with respect to the real machine semantics. The Open Verifier uses *first-order predicates on concrete machine states* as its type of abstract states.

For instance, consider the sequence of assembly instructions from the earlier example that pushes the arguments for a call to `open` onto the stack in the true-case of the `if`:

Assembly code	**Concrete effect**	**Abstract effect**
`movl $1, 4(%esp)`	$\mathbf{r}_{mem} := \mathsf{upd}(\mathbf{r}_{mem}, 4 + \mathbf{r}_{esp}, 1)$	$\mathbf{stack}_{10} := 1$
`movl -8(%ebp),%eax`	$\mathbf{r}_{eax} := \mathsf{sel}(\mathbf{r}_{mem}, \text{-}8 + \mathbf{r}_{ebp})$	$\mathbf{r}_{eax} := \mathbf{stack}_7$
`movl %eax,(%esp)`	$\mathbf{r}_{mem} := \mathsf{upd}(\mathbf{r}_{mem}, \mathbf{r}_{esp}, \mathbf{r}_{eax})$	$\mathbf{stack}_9 := \mathbf{r}_{eax}$
`call open`		

Next to each instruction, we have summarized its concrete effect on a real machine. We treat the memory as an extra register, with values constructed using the standard sel and upd functions for reading or changing a memory cell, respectively (*c.f.*, [HW73]). An extension attached to this program most likely does not care about the details of memory. It is more useful to maintain the same abstraction of a stack that the C language provides. The third column shows an alternate, abstract interpretation of the effects of instructions. Instead of taking the instructions literally, we interpret them as manipulating some infinite array of numbered stack slots.

The extension needs to phrase its abstraction in terms of first order logic. Each instruction transforms a state represented as a list of logical assertions. For example, right before the call to `open` in the true-case of the `if`, the state might include the following assertions:

$$\begin{aligned} \mathsf{CurrentStackSlot}(\mathbf{r}_{esp}, 9) \\ \mathsf{StackSlotIs}(9, string_0) \\ \mathsf{StackSlotIs}(10, 1) \\ \mathsf{StrEq}(string_0, \texttt{"/tmp/results"}) \end{aligned}$$

The CurrentStackSlot predicate relates the stack abstraction with the real machine state. It says that register `esp` is pointing to stack slot number 9. The StackSlotIs predicates records the contents of particular stack slots. Finally, the StrEq predicate was added previously after a call to `strcmp`, based on its post-condition in the security policy.

Open Verifier safety policies are expressed as functions from assembly instructions to logical predicates. A program is considered safe if each time it executes an instruction the predicate assigned to that instruction is provable. For Byte Bouncer, we require that, at any call instruction to a foreign function, the function's pre-condition is provable.

Recall that the relevant part of our example safety policy for the `open` function is as follows:

```
Function open(name, mode)
  Pre StrEq(name, "/tmp/results") || mode == O_RDONLY
```

For the `open` call in the example code, this pre-condition is instantiated as

$$\mathsf{StrEq}(\mathsf{sel}(\mathbf{r}_{mem}, \mathbf{r}_{esp}), \texttt{"/tmp/results"}) \lor \mathsf{sel}(\mathbf{r}_{mem}, 4 + \mathbf{r}_{esp}) = 0$$

Parameters are changed into memory projections, based on the `x86` C calling convention. The extension uses knowledge of its abstraction technique to prove this safety predicate, using the assertions in the logical state as hypotheses. In this case, the extension uses the following key lemma to prove the first disjunct:[1]

$$\frac{\mathsf{CurrentStackSlot}(sp, k) \quad \mathsf{StackSlotIs}(k + n, x)}{\mathsf{sel}(mem, 4 \cdot n + sp) = x}\ \textsc{StackRead}$$

We will not go into the details here of how an extension proves its abstraction sound (*i.e.*, that lemmas like the above hold). The high-level idea is that a successful extension needs to determine what information is worth keeping at which points, and it needs to use that information to prove the safety predicates for all reachable instructions.

In our implementation, we have used a modified version of an extension that we call the PCC extension, first described in [CCNS05], to generate proofs. This extension's strategy is relatively simple. It does four main things:

[1] Some technical details have been elided for clarity's sake, such as an invariant about the well-formedness of memory.

– It provides the abstractions for function call-and-return and disjoint memory regions used for the stack, the heap, and the statically-allocated data;
– It manages some basic bookkeeping that is useful in tracking the execution of programs output by `gcc`, such as information on the contents of particular stack slots;
– It uses loop invariants included in C source files to determine where to introduce abstraction; and
– It uses a proof-generating Prolog interpreter to prove any safety predicates that are encountered.

Actually, the PCC extension does not use "loop invariants" directly, but rather *cutpoint annotations*. A cutpoint annotation designates a point in the code that the code producer expects to be reachable along multiple different paths. Associated with each cutpoint is a description of what parts of program state to remember at that cutpoint; any other state will be forgotten.

For example, suppose we had program code like:

```
mode = O_RDONLY;
while (some_complicated_condition()) {
  name = some_complicated_function();
  fd = open(name, mode);
}
```

Exhaustive verification of all paths through this code may not be possible, depending on the workings of the functions called. However, it's easy to see that the code is safe, because `mode` is always `O_RDONLY`. This is the sort of fact that would go into a loop invariant in traditional program verification. It's also the kind of property we could reasonably expect the code producer to know and use in his justification that the code adheres to the safety policy. In our prototype implementation, we add the following macro use as the first line of the loop:

```
CUTPOINT(FRESHVARS(name, fd));
```

What this means is that, every time this line of code is reached, the contents of the variables `name` and `fd` should be forgotten. As a result, states that would have been considered different will become equal, so that we can check the safety of infinitely many paths. There are a number of other macros that specify ways to forget aspects of program state at cutpoints.

We started from the basic PCC extension and made some improvements necessary to handle more ambitious programs than it had been used with previously. We added support for foreign function calls and their pre- and postconditions. We needed to add support for strings and global and static variables (as compiled by `gcc`), as well as assorted behaviors of `gcc` that we hadn't encountered previously in toy examples. Finally, we developed a Prolog formalization of some useful properties of strings. With these changes in place, the main effort in proving a program safe is in adding appropriate cutpoints to the program source.

To ease the burden on the code producer, these simple annotations, like `FRESHVARS`, can be inferred automatically either by the PCC extension itself or as a separate analysis. In fact, we have written such an analysis that instruments the C source code with cutpoint annotations and the appropriate `FRESHVARS` for our trials. Experimentally, we have not needed more complicated annotations to verify the security policies described earlier (see Section 5 for details).

It is important to remember that Byte Bouncer is not limited to using just this extension. The extension is not a part of the trusted base, and code producers are always free to design their own extensions based on mechanisms particular to their programs. For example, a code producer who programs in Java would be able to provide an extension that could do more automatic reasoning based on the safety guarantees provided by Java. Also, there are some features of C (most having to do with pointers) that the current PCC implementation is often unable to deduce any useful information from. Additional support for these features could be added without needing to change any of the trusted code, since the extension must prove all of its reasoning sound.

5 Case Studies

We began by checking some standard command-line UNIX utilities, like `cat`, with respect to some simple policies (*e.g.*, only read files and no network connections). Then, we experimented with some richer policies, such as those that require simple arithmetic, basic string properties, and use of post-conditions. In this section, we describe larger case studies based on currently available open-source programs implemented in C to evaluate and demonstrate the feasibility of our approach. The examples we chose are perhaps more security critical than the average consumer application in order to "stress test" our approach.

5.1 Pico Server

We have prepared and analyzed the code for the small web server Pico Server, which is available at http://pserv.sourceforge.net/. The source distribution includes about 2000 lines of C code, split among 4 files. We made a few modifications to the code to make it compatible with the current Byte Bouncer implementation. For instance, we elided any code that required non-word sized memory accesses, since the latest Open Verifier version does not support other widths. We refactored small amounts of code dealing with arrays, structures, pointers, and bitwise operations, which we do not entirely handle yet. Besides these changes, we also made a number of semantics-preserving simplifications, some using the CIL library [NMRW02] and some manually.

We have been able to check a number of important properties of the compiled Pico Server object files.

– We allow calling `fopen` for reading by including in the policy file

```
Function fopen(filename, modes)
  Pre StrEq(modes, "r")
```

But we omit any other foreign functions that could together be used to gain access to the file system, so we know that the web server will only perform read-only file system accesses.
– By omitting the `connect` function from the security policy, we know that the web server will never initiate any network connections of its own.
– We also know that Pico Server does not use any other system calls that we did not include in the safety policy (purposely or through negligence), like `remove`.

Our annotation generation program was able to add most of the necessary cutpoint information to the Pico Server source. We only had to spend about an hour adding a few additional annotations. A few more hours were required to change parts of the code that were confusing our verifier, but this problem would not exist in a production version of Byte Bouncer. We think we could improve our annotator enough that the extra annotation work would go away completely, for the complexity of security policies that we used here.

It is worth emphasizing that none of us had ever seen this source code before beginning this case study, and we still managed to come up with certified object files in only a few hours. If programmers have Byte Bouncer in mind while developing their applications, it seems reasonable to expect that they will have a very easy time certifying those applications with policies like the one above.

We check each of the four object files separately, with a combined verification time of about 6 seconds on a 3.2 GHz Pentium 4 with 2 GB of RAM. This time includes both proof generation and proof checking. It would be possible for a user who is willing to trust an extension to speed up the process by not checking proofs. It's also true that the current PCC extension is not optimized for speed. A generic proof-generating Prolog interpreter handles all of the proving. We expect that running time would be much lower in a finely tuned implementation.

5.2 Goldwater

Goldwater, developed by netFluid Technology, is distributed by the FSF under the GNU Public License. Goldwater implements distributed message-based middleware supporting multiple languages, providing services like session persistence, result caching, resource control, load balancing, connection pooling, database connectivity, integration with web servers, web browsers, and wireless device data formats and protocols, XML web services, *etc.* Goldwater has been chosen as the base architecture of the Virtual Remote Server (VRS) project,[2] which is a part of the DotGNU Project.[3] As a larger case study, we chose Goldwater version 1.4.0, framed appropriate security policies for some of its various modules and verified that these policies are satisfied.

[2] VRS aims to create software for a peer-to-peer server clusters for hosting web services. VRS combines the advantages of p2p architecture while still providing benefits of centralized servers, like persistent availability and a common database.

[3] DotGNU is FSF's open-source platform for web services and C# programs.

Module	**File**	**File size** (loc)	**Proof generation** (sec)	**Proof checking** (sec)
`admin`	`goldinit.c`	720	5.80	0.51
`admin`	`bsguardls.c`	315	2.58	0.65
`guardian`	`guardian.c`	1915	4.06	0.64

Table 2. Verification of Goldwater source files.

We selected the `admin` and `guardian` modules of Goldwater. The module `admin` implements Goldwater's administration tool, and the `guardian` module runs at the top level of the Goldwater architecture managing the distributed Goldwater server. We verified the files `goldinit.c`, `bsguardls.c` and `guardian.c` with respect to a specified set of safe foreign functions. As Table 2 shows, proof generation and proof checking are both performed in a few seconds time. With no prior knowledge of the software, verification was accomplished in a few days of work.

6 Discussion

In this section, we discuss some practical considerations for realizing such a system, namely how to make it more usable for both the code producer and the code consumer. We also discuss some other uses for Byte Bouncer that are outside the code producer-consumer scenario we have described.

6.1 Usability for the Code Producer

One of the main concerns with PCC systems is the difficulty of proof generation. A system won't be very useful if code producers find it too difficult to prove that their programs are safe. Some of the work is easily automatable. For instance, we have a tool that adds a cutpoint at the beginning of every loop and automatically annotates it with information on which variables might change across loop iterations. However, we, of course, need heavier weight machinery when programs are safe for complicated enough reasons.

The PCC extension takes the strategy of using a Prolog interpreter to do the drudge work of proving. The code producer is responsible for proving and including with his program additional Prolog proof rules. These might be necessary for proving the program correct, or they might just make verification more efficient. There are generally only a few of these, and it should be possible to reuse them across programs. Program verification experts could create libraries of these rules and distribute them freely, avoiding any need for expertise on logic by the average code producer.

The other main non-trivial component needed by the PCC extension is loop invariants and non-foreign function pre-conditions and post-conditions. These are the hardest to come up with. They are analogous to induction hypotheses in inductive proofs, in that verification can be trivial when they are known and next to impossible when they are not. However, there has been some promising past work on generating loop invariants automatically. For instance, one project modified the BLAST model checker to output loop invariants [HJM+02]. BLAST uses some heavy-weight automated theorem proving machinery to build abstractions for input programs automatically. The nice part about this approach is that much simpler machinery will suffice on the code consumer's end. The consumer only needs to check the inductive proof that the producer and BLAST have spent much effort constructing.

It is also true that much effort that has gone into this project has come from the unrestricted nature of C programs. We chose to study C programs for this project because C is the most common language for security-critical daemons, but our task would have been much easier with a higher-level, type-safe language. Compiled versions of programs in such a language would have much more predictable behavior, making it easier to provide higher level abstractions about them. This adds more evidence for the increasingly popular belief that it's necessary to transition systems programs away from C to achieve practical security goals.

6.2 Usability for the Code Consumer

For any successful security mechanism, one must consider the usability for the consumer, but we suspect that this is a much more tractable problem. The critical issue is displaying policies to the user in as succinct a way as possible, while still enabling the user to make smart security decisions. We think that there are a number of ways to do this well for common types of programs.

For example, Byte Bouncer could come with a number of default policies. Each one would have a descriptive name chosen to be understandable to the average user. For instance, we could have policies named "game," "server," "word processor," *etc.* Mobile programs could come with descriptions of the security policies they are known to satisfy, and the consumer's Byte Bouncer could try to find the most restrictive default policy that implies one of these. If one is found, a simple dialog box could ask the user if he would like to allow the program to run with, for example, "game security settings."

When no default policy is sufficient, it may be possible to find a small delta from one of them. For instance, a given program may behave appropriately for a "game," except that it wants to send data to an unknown IP address. If Byte Bouncer figures this out (a form of "policy inference"), it can make this difference explicit in its query to the user (*e.g.*, "The downloaded software asks to be allowed to read your game configurations file `game.conf`. Do you agree ?"). A system like this might even encourage software vendors to provide more precise documentation, listing every non-standard capability required by a program, along with the reason for its inclusion. This would be a useful consequence in its own right, especially in preventing the installation of unwanted spy-ware.

6.3 Other Uses for Byte Bouncer

Byte Bouncer is also useful as a tool for enforcing software quality. Programmers can make Byte Bouncer checks a regular part of their development cycle. Even if a program is going to be run with more traditional security mechanisms, Byte Bouncer can be helpful in avoiding surprises for end users. When a Byte Bouncer policy models a particular dynamic security mechanism accurately enough, a successful Byte Bouncer run implies that the program being analyzed will never trip that mechanism. Testing-based approaches always leave open the possibility that a program misbehaves in some unexpected case, surprising the user in the middle of execution.

Our system can also be used as a standard tool for checking dynamic instrumentation schemes. For instance, with appropriate security policies, Byte Bouncer can verify that transformations like software fault isolation [WLAG93], StackGuard [CPM+98], and inline reference monitors [ES00] achieve their goals. This double-checks that the potentially complicated instrumentor did not miss an instrumentation point, moving the instrumentor out of the trusted computing base.

7 Related Work

As alluded to throughout the text, our work essentially is an instantiation of Proof-Carrying Code. Prior work in PCC has primarily focused on fixed, low-level policies [Nec97,AF00,CCNS05,HST+02,Cra03,MWCG99] (such as, memory safety), while this work has aimed to lift this idea to higher-level security policies given by the end user. While memory safety is often a prerequisite in order to enforce higher-level security policies, we, in principle, defer the choice of how its enforced to another methodology (*e.g.*, one of the traditional PCC methods or with, say, the Java Virtual Machine). Rather, we have identified kinds of policies that the end user can and would like to specify, as well as the means of enforcing those policies via pre-conditions on foreign function calls.

Our prototype implementation is built on the Open Verifier framework for foundational verifiers [CCNS05], leveraging mechanisms that are common to both enforcing memory safety and our higher-level security policies. As such, we have followed the principle of separating policy from the enforcement mechanism (*i.e.*, trusting only a proof checker and the machine semantics and not, say, the soundness of some high-level type system) from Foundational Proof-Carrying Code [AF00,CCNS05,HST+02,Cra03].

We have explored a common theme in security enforcement—applying static techniques to a problem that has been considered using dynamic or run-time solutions. One dynamic counterpart of Byte Bouncer is Janus [GWTB96], which runs browser helper applications in a environment that checks for dangerous system calls at run-time. The primary advantage of a static approach is that the code consumer can be confident that the code only makes safe system calls prior to running the application. Dynamic and static approaches are not necessarily exclusive. As noted in Section 6.3, one may choose to run Byte Bouncer on the code resulting from instrumentation techniques for enforcing security policies at run-time (*e.g.*, software fault isolation [WLAG93] and inline reference monitors [ES00]).

8 Conclusion

To summarize the design of Byte Bouncer, we evaluate our choices with respect to Saltzer and Schroeder's relevant recommendations for building secure systems [SS75].

- *Economy of mechanism:* The trusted components are simple in design and small in number.
- *Fail-safe defaults:* The user must explicitly specify the foreign functions and on under what conditions they may be called.
- *Complete mediation:* Every call to a function outside the code of the untrusted binary is checked (statically) for possible violations of the user-specified security policy.
- *Open design:* The enforcement mechanism is not secret (and must not be secret for the code producer to generate valid proofs). The security of the system does not depend on knowledge of any secret at all. Instead of a *syntactic* view of security (security that depends on syntactic possession of some information, such as a key), we enforce *semantic security*—the program semantics is analyzed to produce a proof that it abides by the given security policy. This contrasts with "code signing" approaches for mobile code security.
- *Least privilege:* Our approach supports granting privileges to untrusted programs in, essentially, arbitrarily precise increments by the end user. In contrast, the standard privilege-management schemes conflate policy with enforcement mechanism setting up the list of privileges at the system level at the very beginning when the computer system is *designed*.
- *Least common mechanism:* We have minimized the trusted computing base necessary for enforcing the security policy to that of the proof checker, the policy interpreter, and the formalization of the machine semantics.
- *Psychological acceptability:* From the code consumer's perspective, he need only be concerned with the security policies, which are much simpler objects than the program, its execution semantics, or the theorem-prover framework for generating proofs. Writing policies is relatively easy, and requires no knowledge of the behavior or implementation of the untrusted binary. We have also described ways in Section 6.2 how the burden of scripting the policy may be shifted to the code producer. From the code producer's perspective, Byte Bouncer places additional burden on him that can be mitigated to some extent by automated analysis techniques, but in the end, he will only be motivated to take such actions if demanded by code consumers. With the ever increasing virulence of malicious code, it seems plausible that such security enforcement mechanisms shall be demanded by code consumers.

Note that the remaining recommendation by Saltzer and Schroeder, *separation of privilege*, is not relevant in this context, because our scheme does not place trust in any entity based on the syntactic possession of information.

References

[AF00] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-00)*, pages 243–253. ACM Press, January 2000.

[CCNS05] Bor-Yuh Evan Chang, Adam Chlipala, George C. Necula, and Robert R. Schneck. The Open Verifier framework for foundational verifiers. In *Proc. of the 2nd ACM Workshop on Types in Language Design and Implementation (TLDI'05)*, January 2005.

[CPM$^+$98] 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.

[Cra03] Karl Crary. Toward a foundational typed assembly language. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-03)*, volume 38(1) of *ACM SIGPLAN Notices*, pages 198–212. ACM Press, January 15–17 2003.

[ES00] Ulfar Erlingsson and Fred B. Schneider. IRM enforcement of java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, 2000.

[GWTB96] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications: confining the wily hacker. In *Sixth USENIX Security Symposium*, 1996.

[HJM$^+$02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Grégoire Sutre, and Westley Weimer. Temporal-safety proofs for systems code. *Lecture Notes in Computer Science*, 2404:526–??, 2002.

[HST$^+$02] Nadeem A. Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 89–100, Copenhagen, Denmark, July 2002.

[HW73] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica*, 2(4):335–355, 1973.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[Nec97] George C. Necula. Proof-carrying code. In *The 24th Annual ACM Symposium on Principles of Programming Languages*, pages 106–119. ACM, January 1997.

[NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC*, pages 213–228, 2002.

[SS75] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE*, volume 63(9), pages 1278–1308, September 1975.

[WBDF97] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architecture for Java. In *Symposium on Operating Systems Principles (SOSP)*, pages 116–128, 1997.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.