Secure Untrusted Binaries — Provably!

Simon Winwood Manuel M. T. Chakravarty

University of New South Wales
National ICT Australia
{sjw,chak}@cse.unsw.edu.au

UNSW-CSE-TR-0511
May, 2005

THE UNIVERSITY OF
NEW SOUTH WALES

SYDNEY · AUSTRALIA

Abstract

A standard method for securing untrusted code is code *rewriting*, whereby operations that might compromise a safety policy are secured by additional dynamic checks. In this paper, we propose a novel approach to sandboxing that is based on a combination of code rewriting and hardware-based memory protection. In contrast to previous work, we perform rewriting on raw binary code and provide a machine-checkable proof of safety that includes the interaction of the untrusted binary with the operating system. This proof constitutes a crucial step towards the use of rewritten binaries with proof-carrying code.

1 Introduction

Consider the following common scenario: A computer user has obtained a program or program component in binary form and wishes to execute it without the risk of compromising a given security policy; for example, certain files may not be altered. Typically, the user will not have access to the source code and often will not trust the code producer, either due to fear of malicious intent or because of software bugs. Hence, the user needs to *sandbox* the binary, such that the rest of the system is shielded from security violations of the untrusted code.

Sandboxing can either be achieved by operating system abstractions based on *memory protection hardware* or by *software-based fault isolation (SFI)* [13]. SFI is based on *code rewriting*, whereby all potentially dangerous instructions are secured by dynamic checks. This rewriting process is typically performed in the compiler backend [13] or on an assembly representation of the code [11, 3, 8, 6] with some additional constraints, including reserved (sometimes globally so) registers for sandboxing.

In this paper, we propose a novel *hybrid* approach that combines binary rewriting with hardware-based memory protection. Our approach works for raw binary code, where no symbolic information of the source program is available. In contrast to traditional sandboxing by operating systems, we can support fine-grained security policies with standard kernels. Moreover, we provide a machine-checkable proof of safety that includes the interaction of the untrusted binary with the operating system, formalised using Isabelle/HOL. The interaction with the operating system includes memory protection domains as well as fine-grained constraints on permissible system calls. The only other work known to us that includes a machine-checkable proof of safety for sandboxed binaries is McCamant & Morrisett's recent result. Their work extends the original SFI work of Wahbe et al. to CISC processors.

Proof carrying code (PCC) [7] addresses the same scenario by putting the onus of proof on the code producer: Each binary, when being distributed, must be accompanied by a machine-checkable proof of safety. The binary is executed only if this proof is found to be valid.

In practice, few code producers supply the required proofs. Our approach addresses this problem by allowing uncertified components, without requiring the rewriting tool in the trusted computing base. Using our proof of safety for the rewritten binary, the binaries' conformance can be checked without any knowledge of the rewriting process. Alternatively, our methods enable code producers to generate proofs of safety without special compiler support.

Our formalisation is based on a low-level machine model characterising a subset of the Alpha architecture [10], complete with memory protection and an operating system abstraction. On the basis of this machine model, we have formalised the effect of a binary rewriting strategy that introduces

reference monitors enforcing control flow and intercepting all system calls. In particular, we show that the reference monitors in rewritten binaries cannot be compromised and that they detect all security violations; i.e., that the rewritten binary conforms to the security policy enforced by the monitor.

The proof and related lemmas are all machine-checked using the Isabelle/HOL proof assistant. All formal definitions and lemmas in this paper were generated by Isabelle: what you see is what we proved. The proofs are available online at http://www.cse.unsw.edu.au/~sjw/proofs/rewrite.html.

In summary, we claim the following contributions:

- A rewriting technique to enforce security policies involving constraints on system calls in raw binary-only code (Section 2).
- A formal machine-model of a subset of the Alpha architecture, complete with memory protection and an operating system abstraction (Section 3).
- A machine-checkable proof of safety of rewritten binaries (Section 4).

In particular, we do not know of any other hybrid approach that combines OS-mediated hardware protection with binary rewriting, nor do we know of any proof of safety that includes reasoning about protection domains and system calls. We believe that our approach is practical, as most untrusted binaries are distributed as plain machine code and because all modern operating systems—with the exception of some embedded systems—support memory protection. We discuss related work in detail in Section 5.

2 Rewriting, Reference Monitors and Proofs

Continuing the scenario from the previous section, let us suppose that the security policy of our computer user asserts that untrusted binaries can only alter files in the directory `/tmp` and, in addition, write to the special stream `stdout`. As all access to files and streams is via system calls, we need to monitor all system calls issued by the untrusted code.

The operating system (OS) kernel itself already checks that arguments to system calls are within certain limits. However, standard OS kernels support only coarse-grain checks; for example, file access depends on user and group permissions and is not easily changed on a process by process basis for a single user. The argument of other system calls, for example process creation, are even more difficult to monitor. Hence, fine-grained security policies requires reference monitors [9] outside of the operating system.

The obvious place for the monitor is the process that executes the untrusted code. The problem is to ensure that even malicious untrusted code cannot circumvent the monitor. As usual, we ensure this by tracking aspects of the untrusted code's control flow in addition to its system calls.

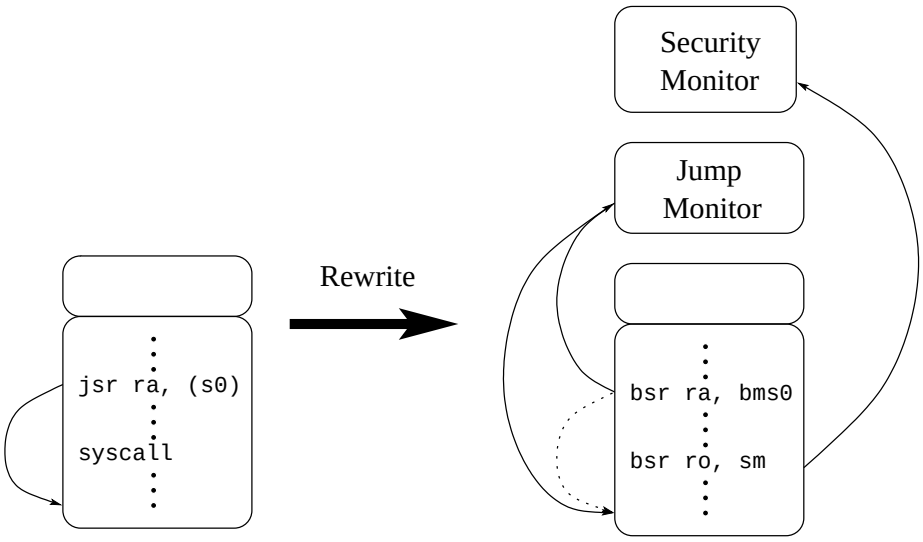

Figure 1: After rewriting, jumps become branches into the branch monitor, system calls become branches into the security monitor

2.1 Securing the Binary

We add two out-of-line reference monitors, which we call the *branch monitor* and *security monitor,* respectively. All jump instructions, of the untrusted code, whose destination we cannot statically determine, are replaced by a branch into the branch monitor. The branch monitor ensures that control is not transferred to an illicit target address and then transfers control to that target. Moreover, all system calls are replaced by branches into the security monitor, which performs the system call if it is admissible. The effect of the rewriting process is illustrated in Figure 1.

The rewriting process simply inspects all instructions and alters indirect jumps (`jsr` instructions on the Alpha) and systems calls as follows:

```
foreach instruction I in binary
   case I in
        syscall    -> bsr  t0, smon
        jsr r, r'  -> bsr  r, jmon(r')
        _          -> I
```

```
jmon (r):
        sll   r, BITS, r
        srl   r, BITS, r
        jsr   (r)
```

Here `jmon(`r`)` is the portion of the branch monitor dealing with jumps via register r. Furthermore, register `t0` is chosen to be a register which may be corrupted by the OS under the calling convention in effect.

Integrity of monitors. The actual enforcement of the security policy is performed by the security monitor, but the branch monitor is needed to pre-

gzip	0.09%	wupwise	9.71%	swim	1.38%	mgrid	1.32%
applu	3.75%	vpr	1.88%	mesa	9.71%	art	-0.10%
mcf	0.18%	equake	4.06%	crafty	18.24%	ammp	-1.35%
parser	8.98%	sixtrack	0.34%	eon	20.41%	gap	17.26%
twolf	7.26%	apsi	0.81%				

Table 1: Increase of runtime in rewritten binaries for a subset of the SPEC CPU2000 benchmark suite on a 600MHz Alpha 21264.

vent the untrusted code to circumvent the checks of the security monitor by jumping in the middle of it. In other words, the branch monitor ensures that entry to the security monitor is only possible at well-defined entry points.

We use hardware-based memory protection to prevent the untrusted code from dynamically altering its own or a monitors code; i.e., all code pages are write-protected. Moreover, we prevent the untrusted code from generating code on the fly in other areas, by using hardware-based memory protection to ensure that only the untrusted code and the monitors are on executable memory pages.

Performance. Rewriting obviously does not come for free, but usually increases the runtime of the untrusted code. However, the benchmarks displayed in Table 1, on the basis of the SPEC CPU2000 benchmark suite, show that these increases are, on average, fairly small.

A restriction of our rewriting method on the Alpha architecture is that the maximum branch displacement is 2MB in either direction, thus limiting the maximum code size.

2.2 Protection Domains

The integrity of our method obviously depends on the availability of hardware-based memory protection. Moreover, in contrast to other approaches to sandboxing, we do not attempt to restrict the targets of load and store instructions of untrusted binaries. Hence, an untrusted binary with its branch and security monitor needs to be placed into its own protection domain.

This is contrary to the purely software-based fault isolation as first popularised by Wahbe et al. [13]. However, three practical considerations support our hybrid approach:

1. In current computing environments, users often need to execute untrusted binaries that are largely standalone programs; i.e., they do not cross protection domains frequently.

2. Recent advances in systems research has demonstrated cross-domain calls with very little overhead [5].

3. Pure software-based approaches often need to rewrite assembly code, whereas binaries are typically distributed as raw binaries that include no symbolic information (not even debugging symbols).

Consequently, we believe that our approach is practical.

2.3 Verifier

Binary code rewriting tools are complex. Hence, they should remain outside of the trusted computing base. However, this requires a verifier that checks at load time that a binary has been appropriately rewritten. In our case, this is simple. The verifier checks that the untrusted code does not contain any `jsr` and `syscall` instructions. Furthermore, all branch targets (of `bsr` instructions) must be within the untrusted code or at appropriate entry points of the branch or security monitor. These checks together are sufficient for the proof of safety (c.f., Section 4.4).

3 The Machine Model

To define a notion of secure program execution, we need a formal semantics of the *instruction set architecture (ISA)* of the processor executing these programs. We achieve this by formalising the relevant parts of the informal definition of the Alpha architecture [10]. Our formal semantics is organised in two stages: Firstly, we define the semantics of each instruction with respect to a simple machine state; secondly, we lift these semantics to a more realistic system state which includes memory protection, exceptions, and an abstract model of the operating system. The following two subsection cover one of these stages, respectively.

3.1 The Base Machine

The *basic machine state* comprises a heap h, a register file rf, and a program counter pc. In addition, we define a decode function from 32bit words to instructions, whose type is shown in Figure 2. This function is lifted to machine states by the $SDecode_1$ and $ADecode_1$ functions ($ADecode_1$ decodes at an arbitrary address rather than the pc).

The evaluation relation is shown in Figure 3. Evaluation is a partial function: For all states s, except those where the current instruction is *Special* there is one and only one next state t such that $s \longmapsto_d t$.

datatype SpecialInstruction = Illegal Word | SystemCall
datatype AOp = AOaddq | AOsubq | AOsll | AOsrl | AOcmpeq | AOcmplt
datatype BOp = BOlt | BOeq

datatype Instruction

= AOp AOp Reg Reg-Or-Lit Reg	*Arithmetic or logical instruction*
| BOp BOp Reg Word	*Conditional branch*
| Br Reg Word	*Branch to address*
| Lda Reg Reg Word	*Load address (add immediate)*
| Ldq Reg Reg Word	*Load word*
| Stq Reg Reg Word	*Store word*
| Jsr Reg Reg	*Jump to address in register*
| Special SpecialInstruction	*System call or illegal instruction*

Figure 2: The Instruction datatype and auxiliary types.

Lemma 1 (Evaluation total 1)

1. If $s \longmapsto^1 t$ and $s \longmapsto^1 t'$ then $t = t'$.
2. If $\forall X.\ \mathit{SDecode}_1\ s \neq \mathit{Special}\ X$ then $\exists t.\ s \longmapsto^1 t$.

We also define a set of well-formed states $\mathit{WfState}_1$, containing only those states which reflect possible Alpha machine states. Evaluation preserves well-formedness.

Lemma 2 (Preservation 1) If $s \in \mathit{WfState}_1$ and $s \longmapsto^1 s'$ then $s' \in \mathit{WfState}_1$.

3.2 Protection, OS Models, and Exceptions

The base system defines an idealised processor: Apart from illegal instructions and system calls, the base system can always transition (Lemma 1). A realistic system imposes additional constraints; for example, valid memory addresses must be aligned. We model this more realistic system state, shown in Figure 4, by lifting the base machine state to include protection domains and operating system state. To favour conciseness, we omit the definition and reference to the notion of well-formed states.

Note that the presented system is more general than required for the reasoning in this paper, as it includes multiple protection domains. We plan to exploit this generality in future work, where we study the interaction between multiple protection domains.

3.2.1 Modelling Operating System Properties

To formalise security policies including constraints on permissible system calls, such as file system access, we need to encode properties of the under-

$$\boxed{S = (h,\ rf,\ pc)}$$

$$\frac{SDecode_1\ S = AOp\ aop\ r_1\ ri\ r_d \qquad v = liftimm\text{-}u\ (bv\text{-}op64\text{-}uu\ (aop\text{-}semantics\ aop)\ (rf\ r_1))\ rf\ ri}{S \longmapsto^1 (h,\ update\text{-}rf\ rf\ r_d\ v,\ update\text{-}pc\ pc)}$$

$$\frac{SDecode_1\ S = Lda\ r_d\ r_s\ off}{S \longmapsto^1 (h,\ update\text{-}rf\ rf\ r_d\ (rf\ r_s \oplus off),\ update\text{-}pc\ pc)}$$

$$\frac{SDecode_1\ S = Ldq\ r_d\ r_s\ off \qquad v = mem\text{-}to\text{-}word\ h\ (rf\ r_s \oplus off)\ 8}{S \longmapsto^1 (h,\ update\text{-}rf\ rf\ r_d\ v,\ update\text{-}pc\ pc)}$$

$$\frac{SDecode_1\ S = Stq\ r_s\ r_d\ off \qquad h' = word\text{-}to\text{-}mem\ h\ (rf\ r_d \oplus off)\ (rf\ r_s)}{S \longmapsto^1 (h',\ rf,\ update\text{-}pc\ pc)}$$

$$\frac{SDecode_1\ S = BOp\ bop\ r\ disp \qquad pc' = (\mathsf{if}\ bop\text{-}semantics\ bop\ (rf\ r)\ \mathsf{then}\ mkbranchpc\ pc\ disp\ \mathsf{else}\ update\text{-}pc\ pc)}{S \longmapsto^1 (h,\ rf,\ pc')}$$

$$\frac{SDecode_1\ S = Br\ r_r\ disp \qquad v = update\text{-}pc\ pc \qquad pc' = mkbranchpc\ pc\ disp}{S \longmapsto^1 (h,\ update\text{-}rf\ rf\ r_r\ v,\ pc')}$$

$$\frac{SDecode_1\ S = Jsr\ r_r\ r_d \qquad v = update\text{-}pc\ pc}{S \longmapsto^1 (h,\ update\text{-}rf\ rf\ r_r\ v,\ bv\text{-}align\ (rf\ r_d)\ 2)}$$

Figure 3: The semantics of the base machine. The term $w \oplus w'$ represents unsigned word addition modulo 2^{64}; *liftimm-u f rf ri* lifts f to the value of *ri*, which may be an immediate or a register; and *bv-op64-uu* returns the result of the give function modulo 2^{64}.

datatype Capability = Read | Write | Exec
types
PD = Domain ⇒ Addr ⇒ Capability ⇒ bool
Exception = Domain ⇒ Addr

record 'a State =

sheap :: Heap	*Memory heap*
spd :: PD	*Protection domain (per domain)*
srf :: Domain ⇒ Regfile	*Register file (per domain)*
spc :: Domain ⇒ Addr	*Program counter (per domain)*
sos :: 'a	*Operating system state*

Figure 4: The lifted machine state. Note that it is parameterised by the type of the operating system state.

$$\frac{\neg\ spd\ S \models spc\ S\ d \uparrow_d Exec}{(S,\ d)\ bad} \qquad \frac{SDecode\ d\ S = Special\ (Illegal\ w)}{(S,\ d)\ bad}$$

$$\frac{SDecode\ d\ S = Ldq\ r_1\ r_2\ off \quad \neg\ spd\ S \models srf\ S\ d\ r_2 \oplus off \uparrow_d Read}{(S,\ d)\ bad} \qquad \frac{SDecode\ d\ S = Ldq\ r_1\ r_2\ off \quad \neg\ bv\text{-}aligned\ (srf\ S\ d\ r_2 \oplus off)\ 3}{(S,\ d)\ bad}$$

$$\frac{SDecode\ d\ S = Stq\ r_1\ r_2\ off \quad \neg\ spd\ S \models srf\ S\ d\ r_2 \oplus off \uparrow_d Write}{(S,\ d)\ bad} \qquad \frac{SDecode\ d\ S = Stq\ r_1\ r_2\ off \quad \neg\ bv\text{-}aligned\ (srf\ S\ d\ r_2 \oplus off)\ 3}{(S,\ d)\ bad}$$

Figure 5: Rules for determining whether a state will raise an exception. There are 3 main causes: illegal instructions, insufficient permissions, and misaligned memory addresses.

lying operating system. However, we would like to do this such that our formal model is not tied to a specific operating system.

We achieve this using Isabelle's *axiomatic type classes*, which support reasoning about the members of a collection of abstract types. An axiomatic type class specifies properties that hold for all instance types of that class. Hence, these properties may be used in proofs about any type of the collection. In other words, these proofs are parametric with respect to the instance type.

System calls are modelled with the *osstep* function and exceptions with the *osexception* function[1]; the operating system type class assumes both functions preserve well-formedness.

3.2.2 Modelling Protection and Exceptions

Protection and exceptions are inter-dependent; hence, we need to handle them together. We define a judgement on states $(S,\ d)\ bad$, shown in Figure 5, which holds when execution of state S should raise an exception, which may be due to protection errors, alignment errors, or illegal instructions.

We model memory protection by a function from protection domains and addresses onto a set of capabilities. We use the judgement $pd \models addr \uparrow_d cap$ to denote domain d has permission *cap* to address *addr* in the protection domain *pd*.

The effect of memory protection is shown in Figure 5: Any attempt at executing an instruction at a non-executable address will result in an exception, as will reading or writing memory without the required permissions.

[1] By modelling system calls and exceptions with functions rather than relations we are assuming a deterministic system.

$$\frac{(S,\ d)\ bad}{S \longmapsto_d\ osexception\ d\ S} \quad \text{(BAD)}$$

$$\frac{\neg\ (S,\ d)\ bad \qquad state\text{-}to\text{-}state_1\ d\ S \longmapsto^1 t}{S \longmapsto_d\ lift_1\ d\ t\ S} \quad \text{(GOOD)}$$

$$\frac{\neg\ (S,\ d)\ bad \qquad SDecode\ d\ S = Special\ SystemCall}{S \longmapsto_d\ osstep\ S\ d} \quad \text{(SYSCALL)}$$

Figure 6: The semantics of the lifted machine. The $lift_1$ function lifts from base states, and $state\text{-}to\text{-}state_1$ does the inverse.

3.2.3 The Evaluation Relation

Next we lift the basic evaluation relation to the complex machine state, as shown in Figure 6, with cases for exceptions and system calls.

Evaluation is total.

Lemma 3 (Evaluation total 2)

1. $\exists S'.\ S \longmapsto_d\ S' \wedge S' \in WfState$
2. If $S \longmapsto_d\ S'$ and $S \longmapsto_d\ S''$ then $S' = S''$.

Memory protection allows us to show that values at non-writable addresses will not change across evaluation.

Lemma 4 (Heap value preservation) If $S \longmapsto_d\ S'$ and $\neg\ (S,\ d)\ bad$ and $\neg\ spd\ S \models a \uparrow_d Write$ and $SDecode\ d\ S \neq Special\ SystemCall$ then $sheap\ S'\ a = sheap\ S\ a$.

4 A Proof of Security

On the basis of the formal machine model, we now outline the proof of safety of rewritten binaries. More precisely, we show that all executions of a program that has been re-written, as described in Section 2, satisfy the security policy enforced by the security monitor.

4.1 Proof Framework

After rewriting, a program has three component: (1) the untrusted code, (2) the branch monitor, and (3) the security monitor. We wish to show properties of each component in isolation; indeed, for this proof, we do not know the details of the security and branch monitors, only that they enforce some security policy. In addition, the security monitor can enforce this security policy only when entered at known addresses, and only for states over which it has control; i.e., those inside the monitor.

$$\frac{S \in \mathit{WfState} \qquad P\ S}{[S] \models_P S \rightarrow_d S} \quad \text{(P-SINGLE)}$$

$$\frac{S \in \mathit{WfState} \qquad P\ S \qquad S \longmapsto_d S' \qquad \sigma \models_P S' \rightarrow_d S''}{S{\cdot}\sigma \models_P S \rightarrow_d S''} \quad \text{(P-STEP)}$$

$$\frac{P \in \Pi \qquad \sigma \models_P S \rightarrow_d S'}{[\sigma] \models_\Pi S \Rightarrow_d S'} \quad \text{(S-SINGLE)}$$

$$\frac{P \in \Pi \qquad \sigma \models_P S \rightarrow_d^! S' \qquad S' \longmapsto_d T \qquad Z \models_\Pi T \Rightarrow_d T'}{\sigma{\cdot}Z \models_\Pi S \Rightarrow_d T'} \quad \text{(S-STEP)}$$

Figure 7: Judgements relating P-sequences and super-sequences.

We thus treat an execution trace as a sequence of *super-states*, each super-state corresponding to execution within a single component. The formalisation is as follows: We split the execution trace into *P-sequences*, where each element in a P-sequence satisfies some membership property. The judgement $\sigma \models_P T \rightarrow_d T'$ asserts that each state in the P-sequence σ (the trace from T to T' in domain d) satisfies the predicate P. A P-sequence represents a super-state.

P-sequences compose into *super-sequences* as shown in Figure 7; the judgement $Z \models_\Pi T \Rightarrow_d T'$ states that the super-sequence Z (from state T to state T' in domain d) consists of P-sequences each of which satisfies a predicate in Π. Moreover, all but the last P-sequence must be *maximal*. A maximal P-sequence is one in which the following state does *not* satisfy the predicate. From a super-sequence one may derive the reflexive-transitive closure, and visa-versa.

Lemma 5

1. If $Z \models_\Pi S \Rightarrow_d S'$ then $S \longmapsto_d^* S'$.
2. If $\forall S{\in}\mathit{WfState}.\ \exists!P.\ P \in \Pi \wedge P\ S$ and $S \longmapsto_d^* S'$ then $\exists Z.\ Z \models_\Pi S \Rightarrow_d S'$.

Finally, we define the concept of a *module*, a datatype with a membership predicate on states and a set of entry points. The intuition is that a program is composed of a number of modules, and the module predicate determines whether a state is within that module. In our proofs each component (the jump monitor, the security monitor, and the untrusted code) corresponds to a module.

The definitions in Figure 8 show the proof obligations for modules. A valid module set must cover all possible states with one and only one module, and define some invariant which at least implies that modules transfer control to other modules via the entry points. If each module preserves the invariant and has the required property, the composition of the modules into a program

$$
\begin{array}{l}
\mathit{module\text{-}pseq\text{-}prop}\ M\ P\ I\ d \equiv \\
\forall\, S\ S'\ \sigma. \\
\quad \mathit{spc}\ S\ d \in \mathit{entry\text{-}points}\ M \wedge I\ S \wedge \sigma \models_{\mathit{in\text{-}module\text{-}pred}\ M} S \rightarrow_d S' \longrightarrow P\ S'
\end{array}
$$

$$
\begin{array}{l}
\mathit{module\text{-}pseq\text{-}inv}\ M\ I\ d \equiv \\
\forall\, S\ S'\ \sigma. \\
\quad \mathit{spc}\ S\ d \in \mathit{entry\text{-}points}\ M \wedge I\ S \wedge \sigma \models_{\mathit{in\text{-}module\text{-}pred}\ M} S \rightarrow^{!}_d S' \longrightarrow \\
\quad I\ (\mathit{eval\text{-}next}\ S'\ d)
\end{array}
$$

$$
\begin{array}{l}
\mathit{module\text{-}set\text{-}ok}\ P\ I\ \mathit{modules}\ d \equiv \\
\forall\, S \in \mathit{WfState}. \\
\quad (\exists! M.\ M \in \mathit{modules} \wedge \mathit{in\text{-}module\text{-}pred}\ M\ S) \wedge \\
\quad (\forall\, M \in \mathit{modules}. \\
\qquad (I\ S \wedge \mathit{in\text{-}module\text{-}pred}\ M\ S \longrightarrow \mathit{spc}\ S\ d \in \mathit{entry\text{-}points}\ M) \wedge \\
\qquad \mathit{module\text{-}pseq\text{-}prop}\ M\ P\ I\ d \wedge \mathit{module\text{-}pseq\text{-}inv}\ M\ I\ d)
\end{array}
$$

Figure 8: Proof obligations for a set of modules.

has that property.

Rule 1 (Compositional rule for modules)
If $Z \models_{\mathit{in\text{-}module\text{-}pred}\ `\ \mathit{modules}} S \Rightarrow_d S'$ and $\mathit{module\text{-}set\text{-}ok}\ P\ I\ \mathit{modules}\ d$ and $I\ S$ then $P\ S'$.

4.2 Exceptions

Next, we discuss the richer exception model required to reason about program behaviour. We have settled on modelling exceptions as a control flow to some address in the program[2]. This address is obtained from the operating system state component by the *exceptos-entry* function.

An alternative model could define exceptions to terminate the program, either using an additional distinguished machine state, or by looping in the same state and so constantly generating exceptions. The approach taken, however, has the advantage of being closer to what is typically implemented by an operating system.

4.3 Weak State Equivalence

We define an equivalence relations on states, shown in Figure 9. This definition reflects the effects of memory protection.

Two states are *weakly equivalent* ($S \sim_d S'$), with respect to a particular domain, if they have weakly equivalent *heaps*, the protection domains are the same (i.e., memory permissions are the same), and the exception entry point is the same. Two *heaps* are *weakly equivalent* ($\mathit{weak\text{-}eqv\text{-}heaps}\ \mathit{pd}\ d\ h\ h'$),

[2]This models the behaviour of some UNIX systems on a signal, for example.

$$
\begin{array}{l}
S \sim_d S' \equiv \\
\mathit{weak\text{-}eqv\text{-}heaps}\ (\mathit{spd}\ S)\ d\ (\mathit{sheap}\ S)\ (\mathit{sheap}\ S') \wedge \\
\mathit{spd}\ S\ d = \mathit{spd}\ S'\ d \wedge \mathit{exceptos\text{-}entry}\ (\mathit{sos}\ S)\ d = \mathit{exceptos\text{-}entry}\ (\mathit{sos}\ S')\ d
\end{array}
$$

$$
\begin{array}{l}
\mathit{weak\text{-}eqv\text{-}heaps}\ \mathit{pd}\ d\ \mathit{h1}\ \mathit{h2} \equiv \\
\forall\, \mathit{addr}. \\
\quad |\mathit{addr}| = 64 \wedge \\
\quad \neg\ \mathit{pd} \models \mathit{addr} \uparrow_d \mathit{Write} \wedge (\mathit{pd} \models \mathit{addr} \uparrow_d \mathit{Read} \vee \mathit{pd} \models \mathit{addr} \uparrow_d \mathit{Exec}) \longrightarrow \\
\quad \mathit{h1}\ \mathit{addr} = \mathit{h2}\ \mathit{addr}
\end{array}
$$

Figure 9: Definition of weak state equivalence.

with respect to a particular domain and protection domain, if each memory location without write permissions but with read or execute permission is identical.

Under normal execution, each state will be weakly equivalent to the following state: Changes to protection domains and exception entry points are only possible via system calls, and memory protection ensures that memory locations without write permissions are preserved.

Lemma 6 (Equivalence preservation) If $S_0 \sim_d S$ and $S \longmapsto_d S'$ and $\neg\ (S,\ d)\ \mathit{bad}$ and $\mathit{SDecode}\ d\ S \neq \mathit{Special\ SystemCall}$ then $S_0 \sim_d S'$.

The following lemma is the main use of memory protection. If two states are weakly equivalent, and an address is executable, then the same instruction will be executed in both states.

Lemma 7 (Decode equality) If $S \sim_d S'$ and $\mathit{bv\text{-}aligned}\ a\ 2$ and $\neg\ \mathit{spd}\ S \models a \uparrow_d \mathit{Write}$ and $\mathit{spd}\ S \models a \uparrow_d \mathit{Exec}$ then $\mathit{ADecode}\ d\ S\ a = \mathit{ADecode}\ d\ S'\ a$.

4.4 The Proof Environment

The initial state of the program is assumed to contain three modules: the untrusted code which has been rewritten, the jump monitor, and the security monitor. The predicates for each monitor are based on the program counter address: We assume some watermark '*ut-top*' between the untrusted code and the jump monitor, and some '*bm-top*' between the jump monitor and the security monitor.

The entry points set for the untrusted module is defined to be all addresses below *ut-top*. The entry points for the other modules are fixed but not defined.

We define an invariant I as

$$I = (\lambda S.\ S_0 \sim_D S \wedge \mathit{spc}\ S\ D \in \bigcup \mathit{entry\text{-}points}\ `\ \mathit{modules})$$

This is the strongest statement we can make about the machine state

after executing untrusted code: Memory protection allows us weak state equivalence, and control-flow rewriting gives us known entry points.

We also fix some policy *syscall-policy* which is a predicate on states that must hold immediately before a system call. The property which must be shown to hold for each module is the following.

$$\begin{array}{l} Pred = \\ (\lambda S.\ can\text{-}exec\ S\ D\ (spc\ S\ D) \wedge SDecode\ D\ S = Special\ SystemCall \longrightarrow \\ \quad syscall\text{-}policy\ S) \end{array}$$

That is, if executing the instruction at that address[3] will not cause an exception, then the security policy must hold for that state.

We assume that the initial protection domain does not allow both *Write* and *Exec* permissions for an address, and that both the exception entry point and initial entry point are within the security monitor. The former is required to handle exceptions in a manner which can be reasoned about, the latter is to allow the security to set up any private state.

The following assumptions about the untrusted code model the effects of the binary rewriting:

- No indirect jumps.

  $$\begin{array}{l} \forall\, addr\ r_r\ r_d. \\ \quad addr <_B ut\text{-}top \wedge can\text{-}exec\ S_0\ D\ addr \longrightarrow ADecode\ D\ S_0\ addr \neq Jsr\ r_r\ r_d \end{array}$$

- No system calls.

  $$\begin{array}{l} \forall\, addr. \\ \quad addr <_B ut\text{-}top \wedge can\text{-}exec\ S_0\ D\ addr \longrightarrow \\ \quad ADecode\ D\ S_0\ addr \neq Special\ SystemCall \end{array}$$

- All branches are within the untrusted code or are to a monitor entry point.

  $$\begin{array}{l} \forall\, addr\ disp. \\ \quad addr <_B ut\text{-}top \wedge \\ \quad can\text{-}exec\ S_0\ D\ addr \wedge \\ \quad ((\exists\, bop\ r.\ ADecode\ D\ S_0\ addr = BOp\ bop\ r\ disp) \vee \\ \quad\ (\exists\, r.\ ADecode\ D\ S_0\ addr = Br\ r\ disp)) \longrightarrow \\ \quad mkbranchpc\ addr\ disp \in \bigcup entry\text{-}points\ `\ modules \end{array}$$

- The last instruction in the text area must be a branch. This restriction is to ensure that the program doesn't simply overflow the watermark. If the next address after the untrusted code is a monitor entry point, this assumption is not required.

[3] *can-exec* predicate holds when a particular address (the program counter in this case) is executable.

$\forall\, addr.$
$addr <_B ut\text{-}top \wedge ut\text{-}top \leq_B update\text{-}pc\ addr \wedge can\text{-}exec\ S_0\ D\ addr \longrightarrow$
$(\exists\, r\ disp.\ ADecode\ D\ S_0\ addr = Br\ r\ disp)$

Finally, we assume that both monitors satisfy the invariant and security properties.

4.5 The Proof of Security

The following lemma is a consequence of the Decode equality lemma (Lemma 7), and one of the principal reasons for our use of protection domains: The value of an instruction in a state reached from the initial state is the same as at the initial state. This implies that properties about the untrusted program assumed in Section 4.4 hold in subsequent states.

Lemma 8 (Decode preservation) If $\neg\ (S,\ D)\ bad$ and $S_0 \sim_D S$ and $S \in WfState$ then $ADecode\ D\ S_0\ (spc\ S\ D) = SDecode\ D\ S$.

The compositional rule for modules requires us to show that the untrusted component preserves the invariant and has the *Pred* property.

Lemma 9 (Invariant preservation)
If $\sigma \models_{in\text{-}module\text{-}pred\ ut\text{-}module} S \rightarrow^{!}_{D} S'$ and $spc\ S\ D \in entry\text{-}points\ ut\text{-}module$ and $I\ S$ then $I\ (eval\text{-}next\ S'\ D)$.

The proof is by induction over the structure of the P-sequence. There are two cases:

P-SINGLE We are then looking at the last state in current module: by assumption, the next state is in another module. We then need to show that the next state is weakly equivalent to the current state, and that the program counter in the next state is in the set of entry points for the next module. These obligations are shown by case analysis over the next evaluation step (evaluation is total, so the next state is always defined).

- **BAD** The next state is an exception, which causes control to be transfered to an entry point in the security monitor.
- **GOOD** From Lemma 4 we have weak equivalence; from the properties of rewritten binaries in Section 4.4 we have that the current instruction must be a branch, and thus is to some module entry point.
- **SYSCALL** It is not possible for untrusted code to perform a system call, so this case is trivially true.

P-STEP In the step case, we have as the induction hypothesis that the invariant holds at the last state. We then need only show that the current state is weakly equivalent to the next state. This is true by Lemma 4.

The next statement asserts that the security policy is respected.

Lemma 10 (Security policy) If $\sigma \models_{\textit{in-module-pred ut-module}} S \rightarrow_D S'$ and $I\ S$ then $\textit{Pred}\ S'$.

This proof is similar to the previous proof, and also proceeds by induction over the P-sequence. With Lemma 8 it is essentially trivial as no system call can occur.

Having shown that the untrusted code preserves the invariant and has the security property, we can now show that the system is secure.

Lemma 11 (Safety of rewritten binaries) If $S_0 \longmapsto_D^* S'$ then $\textit{Pred}\ S'$.

The proof is by Rule 1 using the assumptions about the jump and security monitors from Section 4.4 and Lemmas 9 and 10.

5 Related Work

A number of approaches rewrite at the assembly level, for example Software-based Fault Isolation (SFI) [13] and PittSFIeld [6]. The Naccio system [4] rewrites binaries so that high level resource policies may be enforced. The Gleipnir project [1, 2] uses binary rewriting to ensure control safety. Finally, Prasad and Chiueh [8] present a system based on binary rewriting for preventing buffer overflow attacks.

In the SFI approach, assembly programs are modified so that the address of register-indirect jumps and memory operations are forced to be within a region of the address space. This is done by setting and clearing bits in the top of the address.

An important aspect of the SFI approach is that verifying that a program has been rewritten is separate from the transformation process itself. This results in a smaller trusted computing base.

Although SFI claims to work on binaries, in practice it inserts extra instructions into the program binary, and hence requires symbol information so that the binary can be re-linked. The implementation discussed in Wahbe et. al. [13] uses a modified version of the gcc compiler.

In addition, sharing data between components in a SFI system requires manipulation of the underlying address space. Because SFI restricts components to a single region, it is not possible to address data in another region.

Of the related work listed above, the PittSFIeld and Gleipnir projects are of particular relevance to the work presented in this paper as both verify

the soundness of their approaches. Both projects examine SFI-like systems on the IA-32 platform, a particularly tricky endeavour due to the variable length instruction format; unlike RISC systems, it is possible to branch into the middle of an instruction, and hence execute a different instruction than was statically checked.

The Gleipnir project aims to enforce Control-Flow Integrity (CFI), that is, ensuring that the control flow of a program is not tampered with by an external attacker.

In their approach, a binary's control flow graph extracted by the Vulcan tool [12] is used to generate a set of equivalence classes based on branch targets. These equivalence classes are then assigned unique identifiers, which are inserted before the target code. Whenever a jump is to be taken, the program checks the identifier at the target address contains the correct identifier. If this check fails, then the contents of that register have been tampered with, and thus the attacker can be denied. One point to note is that the Vulcan tool requires a significant amount of information about the binary in order to construct the CFG.

The formal analysis [2] of this approach is a human-checked proof based on a simplified RISC-like processor. The proof assumes that code memory and data memory are disjoint, and that branches into instructions are impossible. They make the interesting assumption that an attacker has control over the heap and then show that CFI holds.

The PittSFIeld project aims to enforce memory and control safety. The basic approach is similar to SFI, however the problem of variable length instructions is addressed by forming instruction groups, each aligned to a k-byte boundary. By ensuring that the bottom k bits of a jump value are cleared (using the SFI technique), this approach removes the threat of jumping into the middle of an instruction at the cost of inserting `nop` instructions into the binary, and hence requiring linking information.

The formal analysis uses the ACL2 theorem prover to model a subset of the IA-32 architecture. A proof of safety is then shown. Their simpler security property (i.e. memory and control safety) is reflected in their proofs: because they do not model a security monitor, they are able to directly induct over evaluation.

6 Concluding Remarks

We introduced a novel hybrid sandboxing technique that combines hardware-based memory protection with binary rewriting. We can support fine-grained security policies with standard operating system kernels and provide a machine-checkable proof of safety that includes the interaction of the untrusted binary with the operating system. The formalisation is based on a machine model that includes protection domains and system calls, which to our

knowledge is a first.

The main limitation of the presented work is that we so far only demonstrated its feasibility for RISC architectures. CISC architectures, and in particular the IA32 ISA, poses new challenges due to its variable-length instructions. We plan to address these challenges in future work.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. Technical Report MSR-TR-05-18, Microsoft Research, February 2005.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. A theory of secure control flow. Technical Report MSR-TR-05-17, Microsoft Research, February 2005.

[3] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *New Security Paradigms Workshop*, pages 87–95, Caledon Hills, Ontario, Canada, September 1999. ACM SIGSAC, ACM Press.

[4] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 32–45, Oakland, CA, May 1999. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press.

[5] Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Herrman Härtig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. Achieved IPC performance (still the foundation for extensibility). In *Proc. 6th HotOS*, pages 28–31, Cape Cod, MA, USA, May 1997.

[6] Stephen McCamant and Greg Morrisett. Efficient, verifiable binary sandboxing for a CISC architecture. Technical report, MIT LCS, May 2005.

[7] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 61–91. Springer-Verlag, Berlin Germany, 1998.

[8] Manish Prasad and Tzi cker Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *USENIX Annual Technical Conference, General Track*, pages 211–224, 2003.

[9] Fred B. Schneider. Enforceable security policies. *Information and System Security*, 3(1):30–50, 2000.

[10] Richard L. Sites and Richard L. Witek. *Alpha architecture reference manual.* Digital Press, third edition, 1998.

[11] Christopher Small and Margo I. Seltzer. MiSFIT: Constructing safe extensible systems. *IEEE Concurrency*, 6(3):34–41, July/September 1998.

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

[13] 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.