

Understanding Precision in Host Based Intrusion Detection

Formal Analysis and Practical Models

Monirul Sharif, Kapil Singh, Jonathon Giffin, and Wenke Lee

School of Computer Science, Georgia Institute of Technology
{msharif, ksingh, giffin, wenke}@cc.gatech.edu

Abstract. Many host-based anomaly detection systems monitor process execution at the granularity of system calls. Other recently proposed schemes instead verify the destinations of control-flow transfers to prevent the execution of attack code. This paper formally analyzes and compares real systems based on these two anomaly detection philosophies in terms of their attack detection capabilities, and proves and disproves several intuitions. We prove that for any system-call sequence model, under the same (static or dynamic) program analysis technique, there always exists a more precise control-flow sequence based model. While hybrid approaches combining system calls and control flows intuitively seem advantageous, especially when binary analysis constructs incomplete models, we prove that they have no fundamental advantage over simpler control-flow models. Finally, we utilize the ideas in our framework to make external monitoring feasible at the precise control-flow level. Our experiments show that external control-flow monitoring imposes performance overhead comparable to previous system call based approaches while detecting synthetic and real world attacks as effectively as an inlined monitor.

Keywords: Anomaly detection, Formal analysis, Program models.

1 Introduction

Over the years, researchers have developed an abundance of host-based intrusion detection systems, utilizing a variety of mechanisms. Most systems, e.g. [9–12, 16, 17, 27, 29], model an application’s normal system call usage and use run-time monitoring to detect attacks that cause behavior deviating from the model. While useful attacks typically require system calls, they provide only a coarse view of a process’ execution. The existence of mimicry attacks [21, 32] that cloak an attack by generating valid sequences demonstrates that attackers may exploit this coarse view. System-call based detectors have another drawback in that they detect attacks well after execution is diverted—at best at the next system call invocation. Monitors verifying system call usage are often implemented as an external process, which eases implementation, debugging, and data protection.

Recently proposed schemes take an alternative approach that detects attacks when they divert control flow. CFI [5], a static analysis based system, efficiently verifies dynamically computed control-flow targets in the program. It guarantees [6] that the execution path will be restricted to the statically generated control flow graph (CFG) of the program, and it can thus detect and stop attacks involving illegal control transfers. In contrast to external monitors verifying a process’ system call use, CFI inlines its checks into the existing code of a program. While inlining complicates monitor development,

debugging, and application to arbitrary binary code, it offers exceptional performance when verifying fine-grained control flow operations.

Speculation about these systems' designs leads to several intuitive conclusions:

- Control-flow based models can provide better attack detection than system-call based models.
- A hybrid model combining control-flow operations and system-call operations better detects attacks than a control-flow based model alone, particularly in the event that static program analysis incompletely identifies control-flows requiring verification code.
- An external monitor cannot efficiently verify control-flow operations.

In this paper, we formally *prove* the first intuition, *disprove* the second, and provide *experimental evidence against* the third. Our goal is to provide clarity to host-based intrusion detection systems research.

In order to understand the strengths and weaknesses or limitations of these anomaly detection schemes, we provide a formal framework to analyze their *precision* in terms of how close they can model a program's normal execution. We first show that for any given system-call sequence based model derived from any program analysis technique (static vs dynamic), *there always exists a more precise control-flow sequence based model*. Such a control-flow based model can precisely match the normal execution behavior of the program from which it was derived, considerably limiting mimicry attacks that plagued system-call based intrusion detectors.

Control-flow sequence based intrusion detectors require the identification of security-critical control flows in a program. A system using static program analysis to identify such control flows may incompletely analyze the program's code due to undecidable problems in static analysis [25]. As a result, a program may contain unchecked control flows. System-call sequence based intrusion detectors face no such shortcoming, as they can completely mediate the system call interface without complete program analysis. Intuitively, we expect hybrid systems combining control flow verification with system call verification, such as PAID [23] to provide better security than control-flow based systems alone. If an attacker breaks out of control-flow checks due to a missed control flow, they can still be detected by the system call checks. Using our framework, we prove that *even if static analysis is incomplete, hybrid models are not more precise than control-flow models*. With appropriate control-flow checks in place, system call checks are redundant and could be removed for model simplification and improved performance.

Finally, we provide experimental evidence against the intuition that efficient enforcement of fine-grained control-flow models can only occur with an inlined monitor. For a fair comparison between the performance overhead of system call and control-flow based approaches, we have implemented an efficient external control-flow based IDS. Using principles developed in our analysis of system-call and control-flow based systems, we apply program transformation to reduce the number of control-flows events exposed to the monitor and improve performance without sacrificing precision. The performance overhead introduced by our detection system, ranging from 1% to 23%, is comparable to previous external system call monitoring. The results also show that our external control-flow monitor can detect a wide range of synthetic and real attacks.

Our current formal framework considers the sequence in which control flows and system calls are executed. As future work, our analysis will incorporate the notion of

data in order to cover approaches that can detect data-only attacks such as program variable or system-call argument manipulation.

2 Related Work

The search for defensive techniques that can detect application-level attacks has led to a rich research area. We consider examples of host-based anomaly detection systems that characterize normal program execution with a language of allowed event sequences. System calls predominantly form the basis of these events, although recent work has developed new models based upon finer-grained control-flow information. Since a primary aim of this paper is to provide illumination of the differences among these model types, we also review previous work in formal analysis of sequence-based models.

Numerous prior systems detect application-level attacks by observing process execution at the granularity of system calls [9, 13–16, 20, 26, 27, 30, 31]. Rather than directly detecting the execution of malicious code, these tools attempt to detect attacks through the secondary effect of the malicious code or inputs upon the system call sequences executed by the process. By allowing attack code to execute, these secondary detectors provide attackers with opportunities to evade detection. Mimicry attacks [18, 28, 32] succeed by appearing normal to a system-call sequence based detector. System call models have grown in complexity to address mimicry attacks, but remain vulnerable because they allow invalid control flows to execute [21]. We note that our paper does not consider non-sequence aspects of system call models, such as characterizations of expected argument values [7, 22].

Control-flow based techniques [5, 34] detect various code execution attacks by verifying destinations of control-flow transfers. Abadi et al. [5] developed Control Flow Integrity (CFI), a recent implementation of control-flow verification. CFI constrains allowed process execution to a model of valid control-flow transfers defined by the program’s static control-flow graph (CFG). CFI uses binary rewriting to place instructions immediately before dynamically computed control-flow instructions for inlined verification of the destination of the transfer. An attacker cannot escape the inlined checks [6] because the static source code analysis or hinted binary analysis can completely identify the set of control transfer points in the program. In this paper, we generalize the idea of CFI to any control-flow based model, including models constructed from training, containing path sensitivity or resulting from incomplete binary analysis.

Given two different classes of sequence-based models, those using system calls and those using control flows, we aim to reason about their attack detection ability. Formal analysis has been previously applied to host-based intrusion detection. Wagner and Dean developed a precision metric called average branching factor (ABF) [31], but this metric is specific to system-call models and cannot be adapted to models of control flow. Chen and Wagner [8] and Giffin et al. [18] use model-checking to find allowed sequences of events in system-call models that execute attack behavior. As with ABF, those tools cannot be adapted to also reason about control-flow models. Gao et al. [12] provided a systematic way of comparing various system call models by organizing them in three axes of design space. This establishes a relation between dynamically and statically constructed system call models, but provides no mechanism to compare system call models with control-flow models. Our formalization not only provides the means to directly compare system-call models with control-flow models, but also provides insight into what effects the precision of control-flow models.

Although the primary intent of this paper is to provide a comparative analysis of system-call and control-flow based models, we additionally consider environments where a hybrid model containing both sets of events may be advantageous. Xu et al. [33] insert waypoints into program code, but these waypoints are not used to verify all computed transfers. PAID [23] inserts notify system calls before indirect function calls so that the monitor can correctly follow indirect control flows. Recent improvements [24] apply this technique to binaries and also incorporate return address checking. We show that hybrid approaches do not provide fundamentally more attack detection capability than control-flow based approaches even in the case of incomplete program analysis.

We also implement an external monitoring based control-flow intrusion detection. We generate the events visible to an external monitor via insertion of null system calls or software interrupts. The Dyck model [17] uses similar code instrumentation techniques. However, the monitor enforcing a Dyck model uses null call events to improve efficiency, not security. In this paper, we use a mechanism similar to null calls for secure exposure of a process' control-flow behavior.

3 Formal Framework for Analyzing Precision

The intrusion detection capability of an IDS is limited by the set of program generated *events* visible to it for modeling and monitoring. In order to compare the attack detection capabilities, it is worthwhile to analyze the relative abilities of recent approaches in terms of how *precisely* they can represent the underlying *normal* behavior of the program they try to enforce. Although our framework enables formal analysis of models comprising any event, we focus on system calls and control-flow transfers. We develop definitions so that they can be applied to both statically and dynamically generated models. We present a *control-flow sequence* based IDS model, which is more precise than any system call sequence based model representing the same valid program execution behavior. This model can precisely represent a program's execution, but requires the exposure of all control-flow events in a program. In Section 4, simplified derivations of this model is used to analyze the precision of practical control-flow based approaches.

Section 3.1 begins with an abstract model of program execution from which we derive all sequence-based models used for intrusion detection. Section 3.2 defines our approach of comparing the precision of different models. We derive system-call based models in Section 3.3 and show that it imprecisely characterizes valid program execution. In Section 3.4, we derive control-flow based models that precisely describe valid execution and consider mimicry attacks in Section 3.5.

3.1 Abstract Model of Execution Sequences

Our abstract model considers the sequence in which code is executed. The smallest unit of executed code is a machine instruction, which can be uniquely identified by its address in memory. Therefore, an execution sequence can be represented as a sequence of addresses from where instructions are executed. Without loss of generality, we consider a coarser *basic block* unit of execution. A basic block is an ordered set of instructions that are executed in sequence as a unit; execution enters only at the start of the block and exits only at the end. The address of the first instruction uniquely identifies a basic block in memory. Basic blocks can also be used to represent high-level statements, making our analysis applicable in the context of both source code and binaries.

main:	placeholder:	E_v (Static Analysis)
B1: if (...)	B6: jmp (*fptr)()	e_1 : B1 B2 B4 B6 B7 B5
B2: fptr = open		e_2 : B1 B2 B4 B6 B8 B5
else	open:	e_3 : B1 B3 B4 B6 B7 B5
B3: fptr = reopen	B7: syscall_1	e_4 : B1 B3 B4 B6 B8 B5
	return	
B4: placeholder()	reopen:	E_v (Dynamic Analysis)
B5: syscall_3	B8: syscall_2	e_5 : B1 B2 B4 B6 B7 B5
return	return	e_6 : B1 B3 B4 B6 B8 B5

Fig. 1. An example program to illustrate control flow and system call based models. The vulnerable function has not been shown.

Fig. 2. The language of valid execution sequences E_v as constructed by a static or dynamic analyzer.

Our abstract models of execution are built on sequences of basic blocks executed by a running program. For a program Pr , let B_v denote the complete set of basic blocks in Pr that can be executed during some valid execution. We use the term *valid* and *normal* interchangeably throughout the paper because from the point of view of an anomaly detector, anything that is deemed normal is considered valid. Figure 1 lists an example program and the basic blocks in its set B_v . Let B_f be the set of all basic blocks that may be feasibly executed in any run of the program. Note that feasible execution differs from valid execution and includes blocks belonging to the program, unknown blocks containing code maliciously introduced into Pr 's address space, and blocks generated by disassembling from the middle of instructions belonging to the program. Clearly $B_v \subseteq B_f$. We next present the abstract models of valid and feasible execution, which are languages over the sets of program points B_v and B_f , respectively.

The Language of Valid Execution: The language of valid execution $E_v \subseteq B_v^*$ contains all sequences of basic blocks from B_v that denote valid execution behavior of Pr . The actual sequences contained in E_v depend upon the algorithms used to compute a program's valid behavior; our framework is general and suitable for any algorithm able to generate E_v . For example, static and dynamic analysis each produce differing characterizations of valid behavior E_v . In the domain of static analysis, different approaches produce models having different sensitivities to program behavior [9]. Dynamic analysis approaches may consider paths that seem valid from the program's static view as invalid. Our framework derives control-flow and system-call sequences from any given E_v . Therefore, our method of comparing precision is orthogonal to the choice of method used to generate abstract model of valid execution E_v .

Figure 2 shows two different languages E_v constructed from typical static analysis and dynamic analysis of the example program. The shaded boxes highlight where the sequences differ. Note that the example program has correlated execution between the direction of the if branch in main and the target of the indirect jump at B7. The E_v constructed from context sensitive static analysis has four possible execution sequences and fails to characterize the correlated execution. However, the dynamically constructed model contains only two sequences because the correlation occurring actual execution carries over to the observed execution sequences.

The Language of Feasible Execution: The language $E_f \subseteq B_f^*$ represents all feasible execution sequences of code in B_f that Pr 's execution can generate. Again, feasible execution need not be valid execution.

The language E_f contains both valid executions and also executions that occur due to an attack. In the example program, any block may appear after blocks B6, B7, and B8 in sequences of E_f because it may be possible for an attacker to change the targets of the control transfers found in those blocks to any address in memory.

The effectiveness of any IDS can be improved by restricting the set of feasible execution paths. Non-writable code (NWC), a standard assumption in almost all systems, restricts feasible execution by disallowing executions that directly modify program code. Non-executable data (NXD) restricts E_f so that executions containing code injection attacks would no longer be feasible behavior. For practical use, the system-call based systems of Section 3.3 require neither NWC nor NXD, but the control-flow based systems in Section 3.4 require at least NWC. Throughout the paper, we assume the usage of NWC.

3.2 Approach of Comparing Precision

We now define how the precision of IDS models can be compared in our framework. For any IDS that models a particular set of events X generated by a program, we can generate the language of valid sequences of such events from the execution language E_v . We will use the notation E_v^X to denote the language containing any sequence of basic blocks that generate a valid sequence of such events. Therefore, any feasible execution of the program that will be considered valid by the IDS is in $E_f \cap E_v^X$. An execution $e \in E_f$ is detected as an anomaly when $e \notin E_v^X$. We will use the following definition for comparing precision of various approaches (illustrated in Figure 3):

Definition 1. *Given the sequence of basic blocks considered valid by two IDSs modeling event categories X and Y from valid executions in E_v are E_v^X and E_v^Y respectively, the former is more precise than the latter if $E_v^X \subseteq E_v^Y$, but not vice versa while keeping $E_v \subseteq E_v^X$ and $E_v \subseteq E_v^Y$.*

From the definition above, IDS modeling X can detect any anomaly detected by the IDS modeling Y . This is because for any feasible execution $e \in E_f$, if $e \notin E_v^Y$ then $e \notin E_v^X$ also. However, more attacks are detectable by the IDS modeling the events X , which are executions in $E_f \cap (E_v^Y - E_v^X)$.

3.3 System Call Sequence Based Intrusion Detection

Our approach for formalizing the precision of system call sequence based schemes is to first derive the language of system call sequences homomorphic to E_v , and then use the inverse homomorphism to identify the language of actual execution behavior allowed by a system call model, which we denote as E_v^S . Let Σ_S be the set of symbols containing all system calls. Without loss of generality, we assume that a basic block contains at most one system call because a block can be subdivided into multiple blocks to meet the requirement. Let $\sigma \triangleleft b$ hold if basic block b contains system call σ . Define the homomorphism $h_s : B_f^* \rightarrow \Sigma_S^*$ as follows. For any $b \in B_f$,

$$h_s(b) = \begin{cases} \sigma, & \text{if } \sigma \triangleleft b; \\ \epsilon, & \text{otherwise.} \end{cases}$$

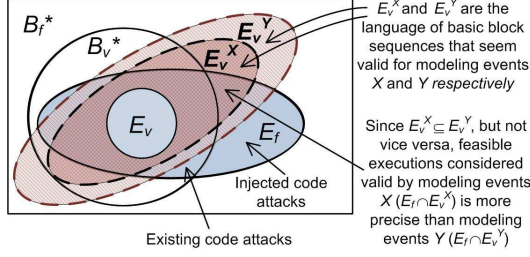


Fig. 3. Illustration of the comparison of precision of two different approaches modeling different types of events.

The language S_v of all system call sequences produced by any valid execution of Pr is then $S_v = h_s(E_v)$. The language containing sequence of basic blocks producing valid system call sequences E_v^S can be found using the inverse homomorphism $h_s^{-1} : \Sigma_S^* \rightarrow B_f^*$ on S_v . It produces the language E_v^S that is less restrictive than E_v :

$$E_v^S = h_s^{-1}(S_v) = \{e \in B_f^* : h_s(e) \in S_v\}$$

The imprecision of system call sequence based approaches can be realized in our framework from $E_v \subseteq E_v^S$. Figure 4 illustrates this relationship along with other languages presented throughout the paper. Using the homomorphism, $e \in E_v \implies s = h_s(e) \in S_v \implies e \in E_v^S$. The contrapositive states that if an executing program generates a system call sequence $s \notin S_v$, then s is generated by a program execution $e \notin E_v$. An invalid system call sequence implies invalid execution. However, the converse is not true; an invalid execution does not imply an invalid system call sequence. This imprecision allows any feasible execution in $E_v^S \cap E_f \supseteq E_v$ to be considered valid. Mimicry attacks [21, 32] that utilize invalid execution exploit this imprecision.

Our framework can be used to derive the known results of Gao et al. [12] indicating that a system call's program counter [27] and calling context [10] improve the precision of system call models by producing a more restrictive E_v^S . However, unless an execution sequence generates a unique system call sequence, E_v^S can never be as precise as E_v . Using control-flow sequences, we can capture the association between consecutively executed basic blocks in order to uniquely represent executions.

3.4 Control-Flow Sequence Based Intrusion Detection

In order to be able analyze the precision of any control-flow based model, in this section we present a *control-flow sequence* based IDS model, which for now, assumes the exposure of all control-flow transfers in a program. We prove that given E_v , we can always derive a control-flow sequence model that provides detection as precise as E_v . This provides an important theoretical result: *for any system-call sequence based model derived using any program analysis approach, there always exists a more precise control-flow based model.*

A control-flow sequence based model characterizes the sequences of control transfer instructions or events that move execution flow from the end of one basic block to the start of another. Let b_S be a special start symbol. Then $\Sigma_C = ((B_f \cup \{b_S\}) \times B_f)$ is the alphabet of control-flow events. Each alphabet symbol represents a pair of the addresses

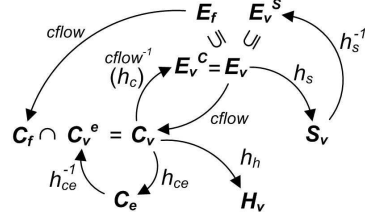


Fig. 4. Relations among various languages in our proofs. Edges indicate functional or homomorphic relationships.

representing a block containing a control-flow transfer instruction and targeted block of control transfer.

A control-flow sequence language is a subset of Σ_C^* . We cannot directly derive a control-flow sequence language from an execution sequence language using a homomorphism because a control-flow transfer depends on two basic blocks—the source and the destination. Instead, we define a function $cflow : B_f^* \rightarrow \Sigma_C^*$ that derives the control-flow sequence from an execution sequence string as follows. For any string $e = b_1 b_2 b_3 \dots b_{k-1} b_k \in B_f^*$,

$$cflow(e) = (b_S, b_1)(b_1, b_2)(b_2, b_3) \dots (b_{k-1}, b_k)$$

In a minor overloading of notation, we also denote the application of $cflow$ to every sentence in a language as $cflow(L) = \{cflow(e) : e \in L\}$. Then, we can derive the language $L_C \subseteq \Sigma_C^*$.

Any language representing a sequence of basic blocks is homomorphic to a language of control-flow sequences:

Theorem 1. *For any language $L \subseteq B_f^*$ there exists a control-flow sequence language $L_C \subseteq \Sigma_C^*$ and a homomorphism $h_c : \Sigma_C^* \rightarrow B_f^*$ such that $L = h_c(L_C)$.*

Proof. Our proof is by construction. We first construct a control-flow sequence language L_C using $L_C = cflow(L)$. We now define the homomorphism $h_c : \Sigma_C^* \rightarrow B_f^*$ as follows:

$$h_c((b_1, b_2)) = b_2$$

The homomorphism h_c is constructed in such a way that we can use it on a control-flow sequence $c = cflow(e)$ derived from some execution sequence $e \in L$, and get the execution string e back, i.e. $e = h_c(c)$. Therefore, $L = h_c(L_C)$. $\square$

We use $cflow$ to derive control-flow sequence languages from E_v and E_f . The language $C_v = cflow(E_v)$ contains only valid control-flow sequences, and $C_f = cflow(E_f)$ contains feasible sequences. Figure 4 illustrates the relations.

This control-flow sequence model precisely characterizes execution. Let E_v^C denote the language of all basic block sequences that can generate valid control-flow sequences, giving $E_v^C = cflow^{-1}(C_v)$. We are going to show that $E_v = E_v^C$. From the definition, $E_v \subseteq cflow^{-1}(C_v)$. Our approach is to show that $cflow^{-1}(C_v)$ is contained in E_v . We first prove that every execution sequence generates a unique control-flow sequence by showing that $cflow$ is one-to-one.

Theorem 2. *Let $e_1, e_2 \in B_f^*$. Then $e_1 \neq e_2 \implies cflow(e_1) \neq cflow(e_2)$.*

Proof. Assume that $cflow(e_1) = cflow(e_2)$. Let $c_1 = cflow(e_1)$. Then $e_1 = h_c(c_1) = h_c(cflow(e_1))$. Similarly, $e_2 = h_c(c_2)$. Since $c_1 = c_2$, $e_1 = e_2$, which is a contradiction. Therefore, $cflow(e_1) \neq cflow(e_2)$. $\square$

From Theorem 2, it is clear that $cflow$ is injective, and $\forall c \in C_v : \exists e = cflow^{-1}(c) = h_c(c) \in E_v$. Hence, $cflow^{-1}(C_v) \subseteq E_v$. Therefore, $cflow^{-1}(C_v) = h_c(C_v) = E_v$. An execution sequence is therefore an anomaly if and only if its control-flow sequence is an anomaly.

Corollary 1. *For any system call sequence model S_v derived from a valid/normal execution E_v , there always exists a more precise control-flow sequence model $C_v : E_v^C \subseteq E_v^S$, but not vice versa where $E_v^C = cflow^{-1}(C_v) = E_v$ and $E_v^S = h_s^{-1}(S_v)$.*

Therefore, any attack that can be detected by a system call sequence language S_v can be detected using the control-flow sequence language C_v , along with more attacks as described in the next section.

3.5 Mimicry Attacks

A mimicry attack [32] is a variant of an attack that achieves the same goal, but can evade detection by an IDS. Better model precision limits opportunities of possible mimicry attacks. For the models of this paper, we set a broad definition of mimicry attacks:

Definition 2. *Given a malicious sequence of events required for an attack, a mimicry attack $\mathcal{A} \in E_f$ is a feasible execution that can achieve the same malicious goal and $\mathcal{A} \in E_v^X$ for E_v^X being the basic block sequence language considered valid by an IDS modeling events X .*

Mimicry attacks on system call based IDS have the freedom of generating feasible executions outside of E_v but in E_v^S to evade detection [21]. Since the control-flow sequence based model is as precise as E_v , any such mimicry attack for which $\mathcal{A} \notin E_v$ can be detected. However, any mimicry attack that switches between valid execution paths, or modifies data only without altering paths cannot be detected. *This is a fundamental limitation of any approach based on execution sequences without data.*

4 Applying Formalisms to Real Intrusion Detection

Our control-flow sequence based model provides the foundation to analyze any control-flow based approach. As presented, the model requires complete exposure and complete history of all control-flow events of a program. Practical control-flow based approaches usually cannot satisfy this requirement due to undecidability problems in program analysis and performance cost. In this section, we simplify our model to analyze the precision of control-flow approaches based on the exposed and covered control-flow events.

CFI [5] only checks the targets of dynamically computed control-transfer instructions, yet it was proven [6] to keep the execution of a program in the statically computed CFG. However, models extracted using dynamic analysis and path sensitive models [34] may need static branches to be exposed for monitoring. Moreover, a control-flow event may be valid or invalid depending on the occurrence of a prior control flow. Hence, the control-flow events that need to be exposed for verification depend on the valid execution model that is being enforced.

We provide a generic framework to derive a simplified yet precise control-flow model that only requires the exposure of a subset of control-flow events. Using this framework, we analyze the precision of CFI in comparison to the system call based models. Our framework also provides insight into derivation strategies for precise dynamically constructed control-flow models. Our goals are different than that of Abadi et al. [6], which proved that the stateless checks provided by CFI were sufficient to constrain program control flow to the static CFG.

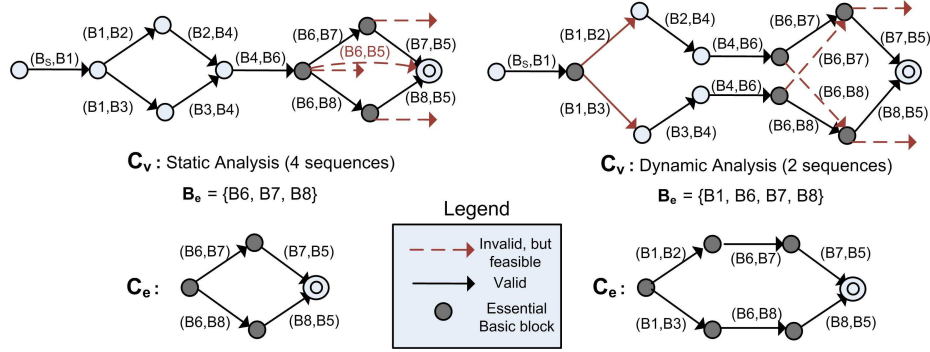


Fig. 5. Simplification of the control-flow sequence language C_v derived from the execution languages given in Figure 2 (The languages are shown by finite state automata).

4.1 Retaining Precision While Simplifying Models

We simplify control-flow sequence models by deriving a simpler language from C_v . We remove control-flow events from C_v that do not help a monitor identify an invalid sequence. If a feasible control-flow event can cause an anomaly or its appearance correlates with another anomalous control-flow event, then it cannot be discarded. We call such events *essential* control-flow events. Any control-flow event emanating from a block that has no essential control-flow events can be discarded without affecting the precision of the model.

Figure 5 illustrates the statically and dynamically constructed control-flow sequence language C_v of the program presented in Figure 1. Invalid but feasible control-flow events emanating from the basic blocks of the program are shown using dotted arrows. Notice that for both models, basic blocks $B6$, $B7$ and $B8$ have invalid but feasible control-flow events. Each control-flow transfer instruction in these blocks uses a dynamically computed target, and can feasibly point anywhere in memory to generate invalid sequences. Additionally, in the C_v constructed from dynamic analysis, the `JUMP` instruction at block $B6$ is correlated with the branch in $B1$. The control-flow event occurring at $B1$ is required to validate the event at $B6$. These control-flow events have to be visible to the model and cannot be removed.

We first define *essential* control-flow events as any of the following:

1. **Anomaly Generating Control-flow Event (AG):** An AG event is the first control-flow event in a sequence to turn a valid sequence into an invalid, but feasible sequence. A control-flow event c is an AG if $\exists \tilde{u}, \tilde{v}, \tilde{w} \in \Sigma_C^*, \forall \tilde{x} \in \Sigma_C^* : \tilde{u}\tilde{v} \in C_v \wedge \tilde{u}c\tilde{x} \notin C_v \wedge \tilde{u}c\tilde{w} \in C_f$ (refer to Figure 6). If c never appears in any valid control-flow sequence, then we call it an *independent* anomaly generator (IAG), which is always anomalous regardless of the events appearing before it. Otherwise, we call c a *dependent* anomaly generator (DAG), which is invalid based upon some previous control-flow events in the sequence. A typical example of an IAG is a feasible control-flow transfer into injected code. In addition, it can be an invalid control-transfer to existing code, such as the event $(B6, B5)$ in Figure 5. Examples of DAG events are function returns that may be sometimes valid and sometimes invalid based on the call site.

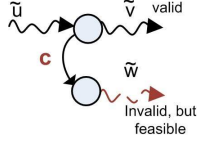


Fig. 6. Illustration of an AG control-flow event c .

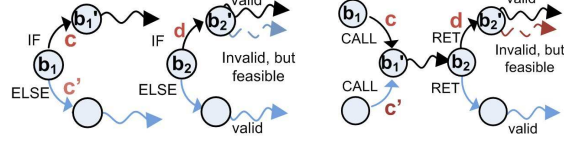


Fig. 7. The two cases where c is an AC event correlated with the DAG event d .

2. *Anomaly Correlating Control-flow Event (AC)*: A control-flow event c is an AC event if its appearance is correlated with a dependent anomaly generator (DAG) event d . Examples are function calls instructions or static branches. More precisely, in order for c to be an AC of d (assume that $c = (b_1, b'_1)$ and $d = (b_2, b'_2)$ here), two conditions must be satisfied. First, in all valid control-flow sequences following the event c , the next control-flow event emanating from b_2 must be d . Second, if c' is another event sharing either the source (e.g. conditional branches) or destination block with c (e.g. function CALL instruction), following c' , if d is the next event emanating from b_2 , it generates an invalid sequences. We show two types of correlation that broadly encapsulates all possible cases in Figure 7. In the dynamic analysis case of Figure 5, $(B1, B2)$ is an AC event correlating with the DAG $(B6, B7)$. If $(B1, B3)$ appears instead of $(B1, B2)$, $(B6, B7)$ generates an invalid control-flow sequence. In case a DAG has multiple AC events, the first one is selected. Our definition can be extended to handle complex cases that involve recursion by incorporating the notion of a stack, and correlating a DAG event with an AC event on top of the stack.

The set of *essential basic blocks* $B_e \subseteq B_v$ contains blocks having at least one outgoing essential control-flow event. The basic blocks in the set B_e are the only ones whose control-flow events need to be exposed for verification. As a result, when a program executes, sequences of control-flow events will be generated from these blocks only.

In Figure 5, the statically constructed C_v has the essential basic blocks $B_e = \{B6, B7, B8\}$ because independent anomaly generating control-flow events exist from them. In the dynamic analysis case, the control-flow events $(B6, B7)$ and $(B6, B8)$ are dependent anomaly generators because they are sometimes valid and sometimes invalid. Since the appearance of $(B1, B2)$ and $(B1, B3)$ correlate to the validity of $(B6, B7)$ and $(B6, B8)$ respectively, they are anomaly correlating control-flow events. Therefore, $B_e = \{B1, B6, B7, B8\}$. Notice that even though block $B1$ contains a branch with static target addresses, it must be visible to the monitor.

Our simplification generalizes to any control-flow model. Unlike CFI, which only considers dynamically computed control-transfer instructions, the set B_e may include control-transfer instructions with static targets if they become an anomaly generating or correlating event. For example, a model enforcing correlated branching would verify the static branches that were correlated. B_e may exclude computed control flows if analysis reveals that an attacker cannot control the destination. For example, an indirect jump reading from a read-only jump table may be safely left unverified.

We define a smaller alphabet $\Sigma_{CE} = B_e \times B_f$ containing only the exposed control-flow events. The simplified subsequence language is derived using the homomorphism

$h_{ce} : \Sigma_C^* \rightarrow \Sigma_{CE}^*$, defined as:

$$h_{ce}((b_1, b_2)) = \begin{cases} (b_1, b_2), & b_1 \in B_e; \\ \epsilon, & \text{otherwise} \end{cases}$$

The simplified model is now $C_e = h_{ce}(C_v)$, which is a language of subsequences of strings in C_v . Again, refer to Figure 5 for the derived C_e of the running example. The model appears to be less precise than the full control-flow model C_v . The inverse homomorphism h_{ce}^{-1} applied to C_e yields $C_v^e \supseteq C_v$. However, this imprecision does not contain any feasible anomalous control-flow sequence:

Theorem 3. *If $C_v^e = h_{ce}^{-1}(h_{ce}(C_v))$, then $C_v^e \cap C_f = C_v$.*

Proof. We first prove that $C_v \subseteq C_v^e \cap C_f$, and then prove $C_v^e \cap C_f \subseteq C_v$. The first part of the proof is straightforward. By definition, $C_v \subseteq C_f$ and $C_v \subseteq C_v^e$. Therefore, $C_v \subseteq C_v^e \cap C_f$.

For the second part of the proof, we show that if $c \in C_v^e \cap C_f$, then $c \in C_v$. The proof is by induction on the length of the string c . Since $c \in C_f$, by definition $c = (b_S, b_1)(b_1, b_2) \dots (b_{l-1}, b_l)$ with $\forall i : b_i \in B_f$. Let $c_s \in \Sigma_{CE}^*$ be the subsequence of control-flow events in c emanating from basic blocks in B_e , i.e. $c_s = h_{ce}(c)$. Since $c \in C_v^e$, $\exists c' \in C_v : h_{ce}(c') = c_s = h_{ce}(c)$.

For the induction base case, we show that some string in C_v begins with (b_S, b_1) . If $b_S \notin B_e$, then (b_S, b_1) cannot be an essential control-flow event. This means that no anomalous sequence can begin with b_S . Therefore, some sequence in C_v begins with b_S . On the other hand, if $b_S \in B_e$, then (b_S, b_1) is in the subsequence c_s and should be the first event in the subsequence. Since $\exists c' \in C_v : h(c') = c_s$ and control-flow events emanating from b_S can only be found at the beginning of a string, c' begins with (b_S, b_1) .

For the induction step, we assume that the $(k-1)$ length prefix of c is also a prefix of some string $c' \in C_v$. We have to prove it for the k length prefix. In other words, assuming that $(b_S, b_1)(b_1, b_2) \dots (b_{k-2}, b_{k-1})$ is a prefix of a valid control-flow sequence, we have to show that the next event (b_{k-1}, b_k) does not induce an anomaly or create a prefix of a sequence outside C_v . First, for $b_{k-1} \notin B_e$, it is obvious from the definition of B_e that no control-flow event emanating from b_{k-1} can create an anomalous sequence. Therefore, the k -length prefix of c has to be the prefix of some sequence in C_v . Suppose $b_{k-1} \in B_e$. This means that (b_{k-1}, b_k) is in the subsequence c_s . The k -length prefix can be invalid only if (b_{k-1}, b_k) is an anomaly generating event. If it is, then we can first reject the possibility that it may be an independent anomaly generator because it cannot be contained in any subsequence of strings in C_v . Therefore, it should be a dependent anomaly generator event. Even in this case, we can prove that it will not create an anomalous prefix of length k . For (b_{k-1}, b_k) to create an anomalous prefix, some anomaly correlating control-flow events should be missing or not in valid order in the $k-1$ length prefix. If that was the case, then the subsequence of essential control-flow events generated by the $k-1$ prefix cannot be the prefix of any subsequence generated by strings in C_v . That contradicts $c \in C_v^e$. Therefore, some string in C_v should have the k length prefix of c . $\square$

Hence, exposing events from B_e and checking with the simplified subsequence model C_e is necessary and sufficient to detect anomalies with the same precision as the comprehensive sequence language C_v with all control-flow events exposed.

Corollary 2. *Checking exposed events from the essential set B_e with the simplified subsequence language C_e is as precise as checking all events with the comprehensive model C_v , which is equivalent to the precision of E_v .*

If basic blocks in B_e are missed, control-flow models become imprecise. The relative precision depend on the covered basic blocks. Models with more exposed control-flow events are more precise.

Corollary 3. *For the same valid and feasible executions of a program, if two control-flow based approaches expose control-flow events from set of basic blocks B_X and B_Y respectively, where $B_e \supseteq B_X \supseteq B_Y$, then the former is at least as precise as the latter, i.e. $E_v \subseteq E_v^X \subseteq E_v^Y$ (basic block sequences considered valid by them are E_v^X and E_v^Y)*

Next, we can state another result that helps reduce the size of the essential basic block set. Restricting feasible execution of a program reduces the set of essential basic blocks without loss of precision:

Corollary 4. *For any program with valid execution language E_v , feasible execution E_f and essential basic blocks B_e , if the feasible execution is constrained such that $E'_f \subseteq E_f$, then the new essential basic block set is $B'_e \subseteq B_e$.*

4.2 Comparing Precision of Practical Systems

Using our framework, we now analyze the precision of several recent host based intrusion detection systems. We first consider models built via static analysis. CFI confines execution in a statically built CFG. Furthermore, it ensures that return addresses are valid by using a protected shadow stack. Suppose that the execution sequences that are paths in the CFG, conforming to proper function call and return semantics, constitute the valid execution language E_v .

We now identify the essential control-flow events. Like CFI, we assume the presence of NWC. As recognized by Abadi et al., any dynamically computed control-flow transfer may feasibly target any basic block. Since they can generate invalid sequences regardless of previous control flow, they are independent anomaly generators (IAG). Returns from functions are dependent anomaly generators (DAG) because they can generate anomalies during an impossible path attack [33]. They are correlated with prior function calls, which are anomaly correlating (AC) events. Therefore, B_e contains all basic blocks that have such instructions. Notice that branch instructions, which have static target addresses are not anomaly generators because both target blocks are valid according to the static CFG.

The blocks in B_e are exactly those covered by CFI. Therefore, if E_v^{CFI} is the basic block sequences considered valid by CFI, then according to Corollary 2, $E_v^{CFI} = E_v$, making CFI the most precise statically constructed sequence based model. Any other system call approach based on static analysis [9, 13, 16, 31] considers the basic block sequences E_v^S as valid, where $E_v \subseteq E_v^S$. Therefore, *CFI subsumes all system call sequence based IDS built on static analysis.*

Dynamic analysis based approaches relying on execution language $E'_v \subseteq E_v$ that are more restrictive than statically constructed models. The system call sequence based models utilizing dynamic analysis [10, 12, 15] recognize basic block sequences $E_v^{S'}$ as valid where $E_v^{S'} \subseteq E_v^S$. When compared with CFI, we cannot say that $E_v^{S'} \subseteq E_v^{CFI}$,

nor can we say that $E_v^{CFI} \subseteq E_v^{S'}$. Our formal framework therefore proves the intuition that neither of the approaches are more precise than the other. Each may detect attacks that the other does not. However, according to Corollary 1, a more precise control-flow model for dynamic analysis exists. Such an approach will be as precise as E'_v , becoming fundamentally more precise than the system call based counterpart because $E'_v \subseteq E_v^{S'}$.

5 Incomplete Analysis and Hybrid Approaches

The precision of control-flow sequence models depends on the exposure of control-flow events in a program. To be as precise as possible, the essential basic blocks at least need to be identified and covered. This is generally straightforward for source code or for binaries with compiler generated hints. However, due to known undecidable problems [25] there is no static or dynamic binary analysis technique that guarantees complete coverage of code for arbitrary binaries. In such situations, an unchecked control-transfer instruction may be exploited by an attacker without being detected by a control-flow sequence based approach. On the other hand, system-call based methods achieve complete coverage of system calls by default because the system call interface can be completely mediated.

A trend toward combining the power of control flows with system calls is evident from PAID [23] with its recent improvements [24]. Intuitively, the advantage of a hybrid approach is that even if an attacker can escape the control-flow verification and execute injected code, a system-call based check should be able to detect invalid system call sequence. However, we show that *hybrid sequence approaches are not fundamentally more precise than control-flow sequence based approaches even in the case of incomplete binary analysis*. One point to note is that PAID considers system call arguments, but since our framework does not consider data, the theoretical results are applicable to sequence based hybrid approaches only.

5.1 The Effect of Incomplete Analysis

In order to help us analyze the effect of models resulting from incomplete analysis, we consider the models that would have resulted if the program could have been analyzed completely. Assume the original definitions of E_v , C_v , B_v and B_e to hold for the models found in the complete case. Let B'_v be the discovered set of basic blocks and E'_v be the valid execution language due generated due to incomplete analysis. Let the essential basic blocks for the incomplete case be B'_e .

The following theorem proves that if the events from the essential basic blocks in the discovered region (B'_e) are exposed by a control-flow based scheme, then the IDS detects any attack that exploits a control-transfer instruction in the undiscovered region.

Theorem 4. *Any feasible execution sequence e that uses an anomaly generating (AG) control-flow event from a basic block in $b \in B_v - B'_v$ is not in E'_v .*

Proof. Without loss of generality, we can assume that e started in the known region of code, i.e. in the blocks B'_v . Since $b \notin B'_v$, prior to any control-flow event emanating from b there must be a control-flow event that transitions outside from B'_v . Such an event is of the form $c = (b_1, b_2)$ where $b_1 \in B'_v$ and $b_2 \notin B'_v$. This event c has to be an anomaly generating event (AG) because it turns a valid sequence invalid. Therefore, $b_1 \in B'_e$ and accordingly is exposed. A simplified control-flow model will therefore detect it as an anomaly, resulting in $e \notin E'_v$. $\square$

Hence, as long as the essential blocks in the discovered region of code are exposed and checked, there can be no undetected attacks that try to exploit unchecked transfers in the undiscovered code.

5.2 Hybrid Models

We can represent hybrid models consisting of both system call and control-flow information in our framework in order to analyze their precision. The alphabet of our hybrid language is $\Sigma_H = \Sigma_C \cup \Sigma_S$, containing both control-flow events and system calls. We can formally describe the derivation as a homomorphism h_h , which has the effects of h_s to add system call information from any basic block, and the effects of h_{ce} to keep a subsequence of exposed control-flow events from blocks in B_h . The homomorphism $h_h : \Sigma_C^* \rightarrow \Sigma_H^*$ is defined as following:

$$h_h((b_1, b_2)) = \begin{cases} (b_1, b_2)s, & b_1 \in B_h \text{ and } b_2 \text{ calls } s \in \Sigma_S \\ (b_1, b_2), & b_1 \in B_h \text{ and no syscall in } b_2 \\ s, & b_1 \notin B_h \text{ and } b_2 \text{ calls } s \in \Sigma_S \\ \epsilon, & \text{otherwise} \end{cases}$$

Using the above homomorphism, the valid sequence model for the hybrid language H_v can be found from the comprehensive control-flow language C_v , by $H_v = h_h(C_v)$. Compared to the pure control-flow and the system-call based models, hybrid models constrain both the control-flow and system call sequences. Therefore, the basic block sequences considered valid by the hybrid model are not less constrained than other two. Assume that the basic block sequences considered valid by a hybrid, a control-flow sequence and a system call sequence models are E_v^H , E_v^C and E_v^S respectively. The following corollary can be very easily derived from our framework.

Corollary 5. *If a hybrid model and a pure control-flow sequence model expose the control-flow events from the same set of basic blocks, then $E_v^H \subseteq E_v^C$ and $E_v^H \subseteq E_v^S$.*

The above shows the relative precision of the three approaches in the general case. However, we will show that in the case that the essential basic blocks are exposed, hybrid models and control-flow models become equal in precision.

It can be proved in a manner similar to Theorem 3 that $C_v = h_{cs}^{-1}(H_v) \cap C_f$ when the essential blocks are exposed, i.e. $B_h = B_e$. Therefore, basic block sequences considered valid by the hybrid model then becomes precise as the valid execution language, i.e. $E_v^H = E_v$. Therefore:

Corollary 6. *If all essential basic blocks B_e are exposed, then a hybrid model is equivalent in precision to a control-flow model, i.e. $E_v^H = E_v^C = E_v \subseteq E_v^S$.*

All that is required to make control-flow based approaches as precise as hybrid models is the coverage of essential basic blocks. We have also seen in the previous section that even for incomplete binary analysis, it is sufficient to cover essential basic blocks in the discovered region of code. Moreover, it is straightforward to identify essential basic blocks in the discovered region of code. Therefore, this shows that control-flow based approaches can be as precise as hybrid models in all cases; hybrid approaches do not have any fundamental advantage over control-flow models. Further research in creating more precise hybrid models may not be fruitful because eventually these systems will become precise enough to make the system call information in the models redundant.

6 Control-flow based IDS Using External Monitoring

Traditionally, system call based IDSes have used an external monitor. CFI uses efficient inlined monitoring to keep the overhead of monitoring at the fine-grained control-flow level low. Although control-flow based methods have been proven to be more precise than system calls, using an external monitor would provide a fair comparison of performance between the two paradigms. We provide evidence against the intuition that an external monitor shifting to this control-flow interface will incur significant overhead. We implement and evaluate a precise control-flow based approach built on static analysis and using external monitoring.

External monitoring has several advantages including easier development and debugging. It can also be easily deployed as a centralized security service. Moreover, it is a more generalized approach that does not rely on tricks to protect memory access to the inlined model or require hardware features such as NXD.

Our external monitor reduces the number of control-flow events that require exposure without losing model precision. We used a run-time program transformation to restrict the feasible executions of a program and hence reduce essential basic blocks. We begin by presenting the implementation details and then demonstrate the validity of the implementation by testing detection of multiple synthetic attacks and real attacks against a collection of test programs. Finally, our performance tests show a surprisingly low cost for external monitoring at the control-flow level.

6.1 Construction via Static Binary Analysis and Rewriting

Our selection of control-flow instructions to model and monitor is similar to CFI. Our model contains a list of valid target addresses for each dynamically computed control-transfer instruction, and a PDA-like stack that stores calling context used to validate the targets of function returns. Like CFI, sequence information is not explicitly required; the stack checks the subsequences of calls and returns. We first construct the static CFG of a program. Then, for each control-transfer instruction that has a dynamically computed target, we use the CFG to identify valid target addresses.

Our system constructs models for dynamically-linked Linux ELF binaries on the x86 architecture. We use *DynInst* [19], a binary analysis and instrumentation library, as our low-level static analyzer. The one-time model construction procedure rewrites the binary program to expose control-flow operations to the external monitor. We use *DynInst* to replace monitored control-flow instructions with single-byte software breakpoints (the `INT3` instruction) that can be securely intercepted by an external monitor.

That monitor limits the program's execution by the model every time the program is subsequently loaded for execution. Using the *ptrace* system call, the monitor intercepts the software breakpoints previously inserted by *DynInst*. For each interception, the target of a control transfer is extracted from the program's context or memory. This method of extracting control-flow information ensures that an attacker cannot pass fake information to the monitor. We implemented the control-flow model itself as a hash table. We key the table on value pairs—a source and destination address for control-flow events. The hash table is sparse with few collisions, providing $O(1)$ average time complexity for lookups. After verification, as *DynInst* had overwritten the original control-flow instructions with breakpoints, the monitor emulates the execution of the clobbered control flow before returning execution control to the monitored process. During execution, our system also intercepts dynamic library loads and updates the model with valid target addresses for indirect jumps that use the GOT.

An external monitor requires context switches into and out of the monitor at every event. We reduce the number of events that the monitor checks by restricting feasible execution of a program (Corollary 4). We use a transformation similar to function inlining. By creating duplicate copies of functions and replacing function call and return instructions with static jump instructions, we remove the necessity of exposing these control transfers to the monitor. In order to reduce code space explosion, we apply a *hot code optimization* that first identifies function calls executed at a high rate at run-time and then performs this transformation. The monitor uses DynInst to alter the code of the monitored process during execution. We ensure that the memory region where the inlined copy resides is write-protected by invoking necessary kernel services.

6.2 Attack Detection

Our approach has the same precision as inlined CFI. We evaluated the attack detection ability of our system after first ensuring the static analyzer and our implementation introduced no false positives for our test programs on normal workloads. We conducted two types of experiments: detection of real attacks against standard Linux programs and detection of various arbitrary code execution attacks against a vulnerable synthetic program.

Our first test evaluated the ability of the external monitor in detecting actual attacks against Linux programs with published vulnerabilities and exploits (Table 1). We ensured that the exploits successfully worked on the vulnerable programs. We then constructed models for each program and used our system to monitor the execution of each program. As expected, the IDS successfully detected every attack before arbitrary code was executed.

Table 1. Detection capability of external control-flow IDS on real applications.

Application	Vulnerability type	Exploit code URL	Detected
imapd 10.234	Stack buffer overflow	[3]	✓
thttpd 2.21	Stack buffer overflow	[4]	✓
indent 2.2.9	Heap overflow	[1]	✓
GnuPG 1.0.5	Format string vulnerability	[2]	✓

Second, we tested the ability of the control-flow based model to detect a collection of injected code and existing code attacks against a synthetic program. The program contains a vulnerability that allows an attacker to write anywhere in data. We created synthetic exploits that modify various code pointers inside the applications’ memory: return addresses on the stack, global offset table (GOT) entries used for locating shared library functions, and function pointers. We tested each control-flow modification with three different classes of targets: injected code, code in the middle of a function, and the entry point of a libc function. Table 2 contains the results of our synthetic attack detection tests.

In all but one synthetic test, our IDS successfully detected the attacks when execution was about to be diverted before the code executed. For the failed test, our IDS missed the attack due to the imprecision introduced in the statically-recovered CFG of the binary code at indirect calls. The target address was a valid function entry point and was thus classified as a normal control-flow transfer by our model. This imprecision demonstrates a shortcoming of static binary analysis that may not be present in static source code analysis or in dynamic analysis.

Table 2. Detection of synthetic tests for various kinds of arbitrary code execution.

Attack Step	Injected	Existing (inside function)	Existing (function start)
Change return address	✓	✓	✓
Modify GOT	✓	✓	✓
Modify function pointer	✓	✓	×

6.3 Performance Impact of External Control-Flow Monitoring

We evaluated the performance overhead on several real-world applications by measuring the execution-time overhead on programs representing both I/O-bound and CPU-bound applications. Table 3 summarizes the results. All timing values represent an average over 5 executions. We first measured each application’s average unmonitored runtime, shown in the results as “Base time”. To determine the time cost of external monitoring of control flow, we then ran the programs with our external monitor. “Monitored time” indicates monitored program execution time. We additionally show the percentage increase in execution time and the percentage increase in program code size due to function body replication during the hot code optimization.

Table 3. Performance results for various applications. Time values are in real-time units.

Application	Base time (sec)	Monitored time (sec)	Time overhead	Inlining space overhead
thttpd	20.40	21.23	4.0%	0.0%
SQLite	55.44	66.04	19.1%	8.8%
gzip	11.03	13.59	23.1%	11.3%
cat	10.06	10.18	1.2%	0.0%

These results show that an external monitor can efficiently detect attacks at the fine-grained control-flow level. Our hot code optimization inlining functions called at high rates effectively balanced the need for fast execution verification with the need to use extra memory responsibly. For example, the I/O-bound applications such as `thttpd` and `cat` incurred a low monitoring overhead and therefore no inlining of code was performed. On the other hand, inlining was crucial for the CPU-bound and function-call-bound program `gzip` for which the crippling performance loss of over 4,000% was brought down to only a 23.1% degradation in speed for an 11.3% increase in space. For comparison, the Dyck model [17] produced a 3% overhead for `cat` for which our system incurs a 1.2% overhead. The earlier model, however, had a 0% overhead for `gzip`, which has a main loop that repeatedly calls functions to compress or decompress data, making only a few system calls. The Dyck model hence can be efficient for this program. Our model instead adds overhead due to the initial control-flow checks and the run-time program transformation needed to optimize away the function calls.

Our control-flow model requires considerably less memory than system call based models such as VPStatic [9] or PAID [23] because it is similar to a single-state PDA. In summary, our IDS ties the power of precise control-flow checks with the convenience of external system call monitoring while keeping performance comparable to previous system-call based approaches.

7 Conclusion

We presented a formal framework for understanding and comparing the attack detection capability of anomaly detection approaches that characterize normal program execution

behavior by modeling and monitoring a set of program generated events. In our principal contribution, we showed that for any system call sequence based approach, there always exists a more precise control-flow based approach. In order to derive more efficient and simplified models, we provided the theory behind selecting essential control-flow events that require exposure. In addition, we proved that control-flow models are more precise even in the case of incomplete analysis, showing that hybrid approaches that include system calls provide only redundant detection. Finally, we used the ideas of reducing essential control-flow events in the program with appropriate transformations in order to make external monitoring at the control-flow level feasible. Our static analysis based approach provides better precision while having performance overhead comparable to previous system-call based approaches.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 0133629. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. We would like to thank Paul Royal for his help in this research.

References

- [1] GNU Indent Local Heap Overflow Vulnerability. <http://www.securityfocus.com/bid/9297/>.
- [2] GnuPG Format String Vulnerability. <http://www.securityfocus.com/bid/2797/>.
- [3] imapd Buffer Overflow Vulnerability. <http://www.securityfocus.com/bid/1130/>.
- [4] httpd defang Buffer Overflow Vulnerability. <http://www.securityfocus.com/bid/8906/>.
- [5] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *Proceedings of ACM Computer and Communications Security (CCS)*, Alexandria, Virginia, November 2005.
- [6] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. A theory of secure control flow. In *International Conf. on Formal Engineering Methods (ICFEM)*, Manchester, UK, Nov. 2005.
- [7] S. Bhatkar, A. Chaturvedi, and R. Sekar. Dataflow anomaly detection. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2006.
- [8] H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, November 2002.
- [9] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, May 2004.
- [10] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, May 2003.
- [11] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, May 1996.
- [12] D. Gao, M. Reiter, and D. Song. Gray-box extraction of execution graphs for anomaly detection. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, October 2003.
- [13] D. Gao, M. K. Reiter, and D. Song. On gray-box program tracking for anomaly detection. In *USENIX Security Symposium*, San Diego, California, Aug. 2004.
- [14] T. Garvey and T. Lunt. Model-based intrusion detection. In *Proceedings of the 14th National Computer Security Conf. (NCSC)*, Baltimore, Maryland, June 1991.

- [15] A. Ghosh, A. Schwartzbard, and M. Schatz. Learning program behavior profiles for intrusion detection. In *Proceedings of the 1st USENIX Workshop on Intrusion Detection and Network Monitoring*, Santa Clara, California, April 1999.
- [16] J. Giffin, S. Jha, and B. Miller. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, California, August 2002.
- [17] J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *Proceedings of the 11th Annual Network and Distributed Systems Security Symposium (NDSS)*, San Diego, California, February 2004.
- [18] J. T. Giffin, S. Jha, and B. P. Miller. Automated discovery of mimicry attacks. In *9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Hamburg, Germany, Sept. 2006.
- [19] J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. In *Proceedings of the Scalable High Performance Computing Conference*, Knoxville, Tennessee, May 1994.
- [20] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference (ACSAC)*, Orlando, Florida, December 1994.
- [21] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the USENIX Security Symposium*, Baltimore, Maryland, August 2005.
- [22] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In *8th European Symposium on Research in Computer Security (ESORICS)*, Gjøvik, Norway, Oct. 2003.
- [23] L. Lam and T. Chiueh. Automatic extraction of accurate application-specific sandboxing policy. In *Recent Advances in Intrusion Detection*, Sophia Antipolis, France, Sep 2004.
- [24] L. Lam, W. Li, and T. Chiueh. Accurate and automated system call policy-based intrusion prevention. In *The International Conference on Dependable Systems and Networks (DSN)*, Philadelphia, PA, USA, June 2006.
- [25] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, Dec. 1992.
- [26] W. Lee, S. Stolfo, and K. Mok. A data mining framework for building intrusion detection models. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, May 1999.
- [27] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the IEEE Symposium on Security and Privacy, Oakland, California*, May 2001.
- [28] K. Tan, K. S. Killourhy, and R. A. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In *Recent Advances in Intrusion Detection (RAID)*, Zurich, Switzerland, Oct. 2002.
- [29] G. Vigna and C. Kruegel. *Handbook of Information Security*, chapter Host-based Intrusion Detection Systems. Wiley, December 2005.
- [30] D. Wagner. *Static Analysis and Computer Security: New Techniques for Software Assurance*. Ph.D. dissertation, University of California at Berkeley, 2000.
- [31] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, May 2001.
- [32] D. Wagner and P. Soto. Mimicry attacks on host based intrusion detection systems. In *Proceedings of the Ninth ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, November 2002.
- [33] H. Xu, W. Du, and S. J. Chapin. Context sensitive anomaly monitoring of process control flow to detect mimicry attacks and impossible paths. In *7th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Sophia Antipolis, France, Sept. 2004.
- [34] T. Zhang, X. Zhuang, W. Lee, and S. Pande. Anomalous path detection with hardware support. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES)*, San Francisco, CA, July 2005.