Software Integrity Protection Using Timed Executable Agents[*]

Juan A. Garay
Bell Labs – Lucent Technologies
600 Mountain Ave.
Murray Hill, NJ 07974, USA
garay@research.bell-labs.com

Lorenz Huelsbergen
Bell Labs – Lucent Technologies
600 Mountain Ave.
Murray Hill, NJ 07974, USA
lorenz@research.bell-labs.com

ABSTRACT

We present a software scheme for protecting the integrity of computing platforms using *Timed Executable Agent Systems* (TEAS). A trusted challenger issues an authenticated challenge to a perhaps corrupt responder. New is that the issued challenge is an executable program that can potentially compute *any* function on the responder. The responder must compute not only the correct value implied by the agent, but also must complete this computation within time bounds prescribed by the challenger. Software-based attestation schemes have been proposed before—new capabilities introduced in TEAS provide means to mitigate the existing shortcomings of such proposed techniques. TEAS are general and can be adapted to many applications for which system integrity is to be tested.

Two types of adversaries to TEAS are considered. First, we address attacks by "off-line" adversaries that attempt to discern agents' functions statically by analyzing their program texts. We then consider "on-line" adversaries, which operate while the agent runs. For off-line adversaries, we show how complexity results from programming language analysis, as well as undecidability considerations, can be used to thwart such analysis by making it impossible for the adversary to correctly decipher all potential agents and reply in a timely fashion. In the on-line scenario, adversaries are difficult to stop in general. We do however present strategies that make it difficult for these adversaries to interpret an agent in a virtual machine and to thereby redirect its actions, for example.

We address the problem of creating large libraries of useful and complicated (and hence difficult to analyze) agents through a new technique of *program blinding*—we hide critical functionality inside randomly generated machine-language programs. We implemented a virtual machine that allows experimentation with this approach. Experiments reveal that blinded agents whose execution conveys important integrity information can be efficiently generated in abundance.

[*]Preliminary version appeared as Lucent Internal Technical Document ITD-04-45822G, Oct. 12, 2004.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: [Language Constructs and Features]; F.2.m [**Analysis of Algorithms and Problem Complexity**]: [Miscellaneous]

General Terms

Security, Verification.

Keywords

Data integrity, intrusion detection, software-based attestation, program analysis, mobile code and agent security.

1. INTRODUCTION

Integrity of computing systems is paramount to their usefulness. Software, hardware, and their interconnection must be trusted to a high degree in order for them to be used in critical and sensitive computations. However, most computing platforms today are relatively devoid of protections against malicious attack where, in particular, resident software is modified or reconfigured. We propose a general challenge-response scheme, based completely on software, to determine and monitor the integrity of software subsystems in a distributed hardware/software architecture.

Similar software-based protection schemes have been proposed before [11, 16], which operate by performing a *fixed* function (checksum) on the state of the system being verified. Such systems however have certain limitations. For example, they have been shown to not be secure [15], or they are not suited for networks where the challenger and the target system are physically far apart or have realistic modern processors with cached data and instructions, or they rely on conjectures that the implementation of the verification function is in some sense optimal in time.

As one example of a system our approach is designed to protect, consider a mobile phone client communicating with hosts in the wireless telephony networks. It is quite possible for the (sophisticated) user of a mobile phone to modify its software. Since a phone interacts inside a network, the host components that communicate with the phone client may be interacting unbeknownst with malicious code. Other examples of scenarios for software integrity protection are: (1) set-top boxes such as cable modems that need to be protected from malicious re-configuration, (2) hardware devices comprised of multiple boards where authentication of the configuration and software of the client boards is desirable to prevent fraud (that enables features not contracted for, for example), and (3) communication or sensor networks in a military scenario where it is critical that client software be intact and not corrupted by an enemy.

Although software clients can, and arguably should, be protected with security hardware (e.g., with a tamper-resistant cryptographic co-processor [18]), they usually are not due to material costs and the additional engineering complexities they impose. Legacy systems already deployed in the field may also require protection *post hoc*—here, an integrity scheme based on software is the only option. Furthermore, many systems are assembled from commodity off-the-shelf components which often contain only minimal hardware security, if any.

Our approach to software-based integrity verification is through *Timed Executable Agent Systems* (TEAS). TEAS typically consists of pieces of computer code—an *agent*—designed to run on a client whose integrity is being ascertained. Since the client will be asked to execute code unknown to it, the agent must be signed by the sending host to signify that it stems from a trusted entity. Upon receipt of a TEAS agent, the client must execute it immediately and return any results the agent produces to the issuing host. Since the agent is an arbitrary computer program that runs potentially with full permissions on the client, it is capable of reading and writing all memory locations and performing calculations on the client state (in the form of checksums, for example). More complicated TEAS instances might perhaps reconcile state modifications made by a prior agent's visit, among many other possibilities.

An important aspect of TEAS is that *the execution period of the agents on the clients is timed*—the output of an agent must be received by the host within a given period of time for it to be valid. As we shall see, this helps prevent potentially malicious code running on the client from expending computation on analysis of the agent and from introducing other overheads, such as those incurred in dynamic interpretation of the agent. The more tightly coupled the target system, the more precise the *a priori* time-bounds for an agent can be. For example in a wireless base-station box where communication is through a tightly coupled backplane, communication as well as execution times can be precisely modeled. On the other hand, in a system that communicates over the Internet, tight bounds on in-transit times may be harder to come by.

In general, TEAS is a probabilistic approach to integrity verification. For example, a single agent may check only a single memory location or compute the hash of a memory region. It is probabilistic in the sense that if a reply is not received in time, it may only signify a network delay and not necessarily a critical integrity violation. It is the collection of many agents operating over a long time period that can establish, with a high degree of probability, that the software running on the client is as intended and expected. TEAS are general and can be adapted to many applications.

A large part of this paper's contribution is a method for generating computational agents randomly in order to populate efficiently large libraries of useful agents. Manually written agents, interspersed with random machine-generated agents, both further obfuscated to create additional semantically equivalent variants, provide a large obstacle that adversaries would need to "learn" in order to combat TEAS protections effectively.

We discuss related work in more detail in §6. Here we mention briefly the previous systems most relevant to TEAS. One of the first software-only schemes to verify the integrity of computer systems was Kemell and Jamieson's *Genuinity* system [11]. Genuinity essentially uses a checksum of pseudo-random virtual memory addresses, machine-specific register values, and time to verify integrity. Genuinity's trusted host also computes the checksum, and if the client being tested calculates the same checksum as the trusted host and returns it "quickly enough," the client software is considered genuine, since it is assumed that there would be no fast simulation. However, Shankar, Chew and Tygar [15] present such simulation (with running time below the 35% slowdown allowed by Genuinity), by mounting what they call a *substitution attack* on the system: Imposter code that sits in the client's checksum page available memory, making sure that correct values are incorporated into the checksum. The work of Shankar *et al.* shows that there are "holes" in Genuinity, making some assumptions about memory utilization, that may or may not hold in practice. Central to their attack is the knowledge that Genuinity is computing this checksum. Agents in TEAS can of course compute similar checksums, but are not restricted to one known function.

Seshadri *et al.*'s SWATT system [16] also uses a checksum of a pseudo-random memory traversal to probabilistically ascertain that memory has not been corrupted by malicious code. In contrast to Genuinity, SWATT focuses on small embedded microcontrollers, with fixed processor speeds and (small) memory sizes. Importantly, SWATT relies on an extremely tight coupling between the trusted host and the target system to work properly. SWATT also requires knowledge of the entire state being checked—or, equivalently, requires the transmission of the client's state to the trusted host as part of the verification. SWATT's checksum needs a large number of memory accesses ($O(n \log n)$, where n is the size of the memory) in order to guarantee the checking of all memory locations with high probability. Additionally, the code implementing the checksum must not admit further optimization in order to exclude malicious interpretation of the checksum instructions; it is known that—at least for non-trivial pieces of code—determination of this property of the code is impossible.

In similarity with Genuinity and SWATT, TEAS does not provide a provably correct solution to software-based verification. It is however more practical than SWATT in some dimensions and safer than Genuinity because TEAS allows for the challenges (agents) to do perform more complex and diverse tasks than a checksum—TEAS adversaries must adapt in real time to TEAS' ever changing verification functions. TEAS' diversity makes it more difficult for the adversary to resist detection in all circumstances. Additionally, TEAS does not require knowledge of the entire state of the system being verified, Furthermore, against some adversaries TEAS uses a technique that we call *program blinding* to obfuscate the functionality of the agent, and we rely on the complexity of program analysis in order to prevent the adversary from deciphering it. To our knowledge, this is the first time that the complexity of program analysis is used for intrusion detection.

Organization of the paper. The remainder of the paper is structured as follows. In §2 we describe our computational/network model and give prerequisite definitions. We then address integrity protection against the two classes of adversary types—on-line and off-line—in §3 and §4, respectively. We present some applications and practical considerations for TEAS in §5 and a review of related work in §6. We conclude the paper with a summary and directions for further research in §7.

2. DEFINITIONS, ASSUMPTIONS AND SYSTEM REQUIREMENTS

This paper concerns collections of computational nodes in a (possibly *ad hoc*) network. The nodes comprise two types of entities: secure hosts, and possibly insecure—in the sense of being subject to malicious attacks—clients.

As mentioned in §1, the goal is for host nodes to establish the integrity of the client systems. We assume that the clients execute computer programs specified with the RAM model of computation. The RAM model corresponds well with the architectures of contemporary microprocessors and DSPs such as those produced by many manufacturers. Load-store and memory-based instruction sets, as well as stack architectures, are readily captured by the RAM model. In this paper we consider hosts and clients that are uniprocessors; we use C, $[C]$ = cycles/sec, to denote the CPU rate of such processors. We further assume that the clients do not possess any tamper-resistant hardware (containing, for example, a secure processor to perform cryptographic operations). The memory of a client consists of four areas: code, data, unused memory, and program stack. Naturally, parts of the data area may be unused at different points in time. We will use M to denote the size of a client's memory, measured in number of words.

Regarding communication, we assume that the nodes in the system, especially the hosts, have a fairly accurate estimate of the transmission delays in the network. This is modeled by hosts' knowledge of the available bandwidth on the links or channels connecting them to the clients. We will use B, $[B]$ = bits/sec, to denote the available bandwidth on a given link.[1]

Adversary model. The aim of our constructions is to provide defense against client nodes being corrupted and/or taken over by an attacker, which we will refer generically to as *the adversary.* In the applications we have in mind, the adversary will typically be *a program* running in the client's memory, such as a computer virus, or code implanted by the enemy or a malicious insider. Since the adversary will be executing at the client, it will be bounded to the client's computational power. More specifically, and as in [16], we require that the adversary makes *no changes* to the client's hardware, such as increasing its memory or increasing its clock speed.

Therefore, attacks are all software-based, and can range from the adversary simply making modifications to the client's configuration or state, to actively controlling the client. Our defense strategy against these attacks will consist of sending programs ("agents") to perform various tasks at the client. In order to go unnoticed, the adversary will have to provide the correct output, and—importantly—in a timely fashion. Thus, we divide adversaries into two classes, according to the type of analysis that they will perform on such incoming programs;

- *Off-line adversaries*: In this class we assume that the adversary controlling a client will try to analyze the incoming programs *without running them.* Recall that in *static analysis* of programs [12, 1], programs are analyzed in isolation, without inputs and without the state of the machine where they will run. An off-line adversary will perform a similar type of analysis, except that it might also have access to inputs and state of the client. We let $\mathcal{A}_{\mathit{off}}$ denote the class of off-line adversaries.
- *On-line adversaries*: In this class, we assume that the adversary controlling a client will also be able to run the incoming programs. We use $\mathcal{A}_{on}$ to refer to this class of adversaries.

[1]Estimation of measures such as computation rate and bandwidth can be performed empirically.

We call a node (client) that has been attacked by an adversary *corrupted.*

Since we assume that an adversary will execute at a client whose computing power is known, computation steps can be easily transformed into absolute time.

Timed Executable Agent System (TEAS). At the core of our constructions is a general form of a challenge/response system, where arbitrary challenges—or *agents*—are issued between the various nodes in the network. We call the node issuing the challenge (typically a secure host) the *Challenger*, and the node being queried (typically a client) the *Responder*. Examples of tasks that such agents might perform include start executing at an arbitrary point of *Responder*'s state, perform some (random) computation including the content of memory locations or the state of *Responder*'s stack, perform modifications (e.g., a random permutation) of the state so as to disable a potential attacker controlling *Responder*, *etc.* Besides the result expected by *Challenger* from the execution of the agent at the target node, an important quantifiable side effect is the *time* it takes the agent executing at *Responder* to provide the (correct) result. We now provide a more formal definition of the agent system.

An $(\epsilon, \mathcal{A})$ *Timed Executable Agent System* (TEAS) is defined by the following two (probabilistic) algorithms, running on the node acting as *Challenger*:

1. The *agent generation algorithm* Tgen, which on input of environmental parameters (the link's bandwidth B and *Responder*'s CPU rate C), outputs a TEAS instance $T = \{(P_1, o_1, t_1, \pi_1), (P_2, o_2, t_2, \pi_2), \ldots, (P_k, o_k, t_k, \pi_k)\}$, where P_i, $1 \leq i \leq k$, is a program (agent) to be executed at *Responder*, o_i is its expected output, t_i is its expected transmission and execution time, and π_i is its *patience* threshold.

   Let $|P_i|$ denote the size of the ith program, and let $D(P_i)$ denote the number of dynamic instructions executed during P_i's execution. Then t_i will typically be computed by Tgen as
   $$t_i = \frac{|P_i|}{B} + \frac{|o_i|}{B} + \frac{D(P_i)}{C}. \qquad (1)$$

2. The *agent verification algorithm* Tver, which takes as input T and the list of pairs $\{(o'_i, t'_i)\}_1^k$, where o'_i is the result provided by agent P_i after executing at *Responder*, and t'_i is the elapsed time since the agent was sent to *Responder* until the result was received, as measured on *Challenger*'s clock, respectively, and outputs $v \in \{\text{'OK'}, \text{'}\neg\text{OK'}\}$.

We require that if *Responder* is corrupted by an adversary in adversary class $\mathcal{A}$, then the probability that Tver will output 'OK' is less than ϵ. On the other hand (and to simplify), if *Responder* is not corrupted, then Tver should not equivocate and always output 'OK'.

Program analysis background and requirements. We define a *halting program* to compute a function from a set of input values to a set of output values in a finite number of computation steps. We consider two programs as syntactically equivalent if their texts are the same; *i.e.*, the bit strings representing the programs are identical. We use *txt*(P) to denote the text of program P. Two programs are *semantically equivalent* if they have the same input/output relation.

```
{     a = 0;
      while (a < 10) {
            b+=M[a];
midloop:
            a++;
      }
...
      if(x > 0)
            goto midloop;
...   }
```

Figure 1: *Sample program containing unstructured control flow (goto jump back into a loop body) that leads to irreducible flow graphs, which in turn give rise to expensive analysis times.*

The approach we propose in this paper to stymie off-line adversaries, as described above, is to use the algorithmic complexity of program analysis (see [12] for background) or the undecidability of most non-trivial program properties (see [10]). The automatic analysis of a program's text, typically for optimization purposes, is almost as old as programming languages themselves. Comprehensive treatments of this field are given by Muchnick and Jones [12] and by Aho, Sethi and Ullman [1]. The analyses and techniques we draw on can be found in these standard works. Note that here we do not consider new probabilistic analysis techniques, such as the recently proposed *random interpretation* analysis [?].

In a nutshell, the behavior of a program P in terms of the program's output values as well as of any *side effects* that the program may perform may be determined in part through some *global data-flow analysis* on the program. Given *txt*(P), some of the following tasks must be performed in order to deduce statically its operation:

- extract P's control flow graph (CFG) G_P;
- convert G_P to a reducible flow graph G'_P (see below);
- perform global data flow analysis on G_P (or G'_P).

In the best case the extraction of the CFG has time complexity $\Omega(n)$, where n is some static measure of the size of P, such as the number of instructions it contains, or the number of assignments it has; but in the case of certain types of branches, this complexity rises to superlinear—$\Omega(n^2)$ or even higher [7]. Also, the resulting CFG may or may not be *reducible* (*cf.* [1, 12]) depending again on the presence of branches into loops. Figure 1 contains a program fragment that, due to the unstructured goto into a loop body, gives rise to an irreducible flow graph. As we shall see in §3, a

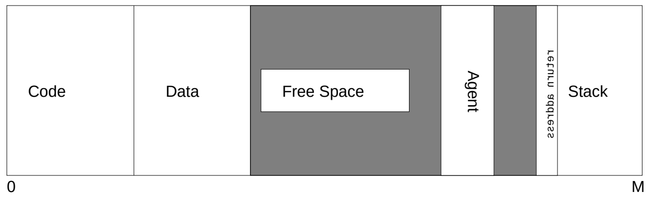

Figure 2: *Memory map of Responder, with agent executing.*

goal of agent generation is to make analysis difficult—this can be done by introducing irreducible flow constructs. Conversion of an irreducible graph to reducible one is possible via so-called *node-splitting*, which however can increase the program size (and hence its CFG size) exponentially [1, 12]. Global data flow analysis with the iterative algorithm on an irreducible CFG has complexity $\Omega(n^2)$, but can approach linear time on a reducible CFG using alternate algorithms.

System requirements. Common computing platforms such as PCs or servers suffice to implement the *Challenger* platform since it need only periodically issue challenges and perform bookkeeping of its client *Responders*. Unlike the *Challenger*, a *Responder* must meet special system requirements regarding the access and use of its resources in order to be used in a TEAS system. In particular, it must permit the execution of arbitrary code, albeit authenticated by a trusted *Challenger*. However, since agents for TEAS are constructed in conjunction with the particular application in mind, we assume that the TEAS agents are not going to harm the application. Agents automatically generated are constructed in such a way as to adversely affect a *Responder's* state (Section 3.3). The authentication procedure (say, public-key based) guarantees *Responder* that it is getting a correct agent and does not introduce a new vulnerability into the system.

In addition, the *Responder*'s operating system must allow an agent full and uninterrupted access to the machine for the duration of the agent's execution—since agents would be typically small and have low time complexities, this is not an issue. In practice, this means that *Responder* must disable interrupts (*etc.*) to prevent external or internal events such as IO or time-slicing from the OS's process scheduler. A *Responder* must also allow an agent access to all memory.

Another requirement is that *Responders* are provisioned to receive and execute agents. A *Responder* therefore requires a (small) portion of free memory in which to place the agent as well as the trivial software machinery for doing this placement and for launching execution of the agent. This is illustrated in Figure 2. More complicated schemes might vary the locations of memory used for this; since agents are general programs, they can modify the software, memory locations, and parameters used for subsequent agent invocations on a given *Responder*.

We assume that existing network transmission techniques are used to acknowledge the correct receipt of an agent, so that in scenarios where operation of an agent depends on the operation of a previously sent agent, the correct order of operations will be applied. That is, an acknowledgement based transmission scheme might be used where transmission is retried until an agent is correctly received. (Note that a retry might send a different agent than in the failed transmission—it must only be in sync with the functions of the other cooperating agents.)

Although it is unlikely that off-the-shelf operating systems, such as ones common on PC's, are adequate for TEAS due to the aforementioned requirements, the customizable real-time operating systems used in many devices such as mobile phones or proprietary computing appliances like wireless basestations are sufficiently customizable and low-level to admit a TEAS security approach.

3. INTEGRITY PROTECTION FROM OFF-LINE ADVERSARIES

The approach we propose to combat off-line adversaries is to use the algorithmic complexity of program analysis (see [12] for background) or the undecidability of most non-trivial program properties (see [10]). We first briefly elaborate on the latter, and then turn our attention to the former.

3.1 Undecidability-based protection

A simple argument conceptually against the feasibility of off-line analysis of an agent is the undecidability of non-trivial program properties as given by Rice's theorem [10]. Namely, properties such as whether generally a program halts or how many steps it takes are non-computable so they cannot (for all programs) be determined by an off-line adversary.

Take for example the non-trivial program property of computing the number of instructions a TEAS agent executes. An agent could incorporate this dynamic value into the result it returns to the *Challenger*. The number of instructions executed can be trivially computed by the agent as it runs, but is non-computable *a priori*. Any amount of off-line analyses cannot exactly determine such an undecidable property.

Unfortunately we do not yet know of automatic methods for generating agents with given undecidability properties (e.g., number of instructions executed). We therefore turn our attention to complexity-based protection schemes where we can present methods for automatic agent creation.

3.2 Complexity-based protection

Effectively, an off-line adversary is a *program analysis*-based adversary, i.e., one which attempts to analyze the flow of an agent and tries to deduce the result(s) of the agent using this analysis. By doing this, the adversary hopes to discern, for example, which memory locations the agent accesses. In the case where the adversary needs to "protect" or "disguise" certain locations because they are being used by the adversary toward a malicious end, such an analysis can help pinpoint which locations must be temporarily "restored" to their proper values before executing the agent. (Note too that address values may be computed dynamically in the agent, so non-trivial analyses are necessary to determine all critical locations.) Furthermore, the analysis must infer also the function that the agent computes on its inputs. As pointed out in §2, program analysis has inherent time and space complexities which we will use to force the adversary to expend computation, and hence time, on the analysis task which makes it harder for the adversary to return the desired value(s) to the *Challenger*.

To combat off-line adversaries using program analyses in such a manner, we construct our agents so that the time complexity of performing the analysis causes the adversary to violate the TEAS time frame for receiving a response. In particular, we utilize the superlinear time complexity of performing program analysis (e.g., $\Omega(n^2)$ in the case of *iterative data-flow analysis* [12]).

Example: More concretely, consider a system with the following bandwidth and computation parameters: $B = 10^6$ bytes/second and $C = 10^9$ cycles/second and on average the processor completes one instruction per cycle. Suppose some TEAS agent is of size 10^3 instructions where each instruction is four bytes in size. Furthermore, suppose this agent to have linear dynamic runtime therefore requiring at least $10^3/10^9 = 0.000001$ seconds to complete on the client. If the agent's result fits in four bytes, the total time for communicating the agent from the host to the client and returning the client's result to the host is bounded below by $4 \times 10^3/10^6 + 4/10^6 = 0.004004$ seconds for a total communication and computation time of $t \geq 0.004005$. Now, if the adversary must perform at a minimum a $\Omega(n^2)$ analysis with an implied constant ≥ 1, the adversary must expend an additional (at least) 0.01 seconds of computation time giving it a time of $t' > 0.014005$, therefore dominating t by a factor of (roughly) 3.5—a significant amount that should be readily detectable in practice. By increasing the size of the agent further this gap can be made arbitrarily large.

This example requires additional commentary. First, it is necessary that the mix of agents sent to the client contain agents with fast (linear or constant running times) in addition to more complex agents (perhaps with quadratic or cubic execution time). This will ensure that for some agents the analysis will dominate the patience threshold (π) for those agents; this point can be handled by judicious agent construction (*cf.* §3.3). Second, it must be possible to guarantee that for many non-trivial agents, an adversary's analysis time be asymptotically larger than the agents running time.

We now show how common analyses can be made to require superlinear (in the size of the agent) time—this again is possible through careful agent construction.

```
     statement                        defines    uses
{    a = 10;                          a
     b = a + 1;                       b          a
     while (a-- > 0) {                a          a
           c = a+b+M[a];              c          a,b,M[a]
           b++;                       b          b
     }
     return c; }                                 c
```

Figure 3: *Results of a* def-use *analysis on a sample program. Program statements are on the left, lists of variable definitions in the center column, and lists of variable uses in the rightmost column. The def-use lists can be automatically inferred by a def-use program analysis.*

An off-line adversary to TEAS must determine the critical locations inspected by an agent, the function it computes, and also any *side-effects* the agent may perform. A side-effect is a modification to a program's state that does not directly affect the result value but can influence the subsequent computation by the client or of future agents. An off-line adversary must therefore perform some *global data-flow analysis* (*cf.* §2) on the agent in question in order to determine this.

As mentioned above, it must be the goal of the agent writer to make analysis asymptotically more expensive than execution, for at least some subset of agents. Here are two ways one might do this:

1. Make CFG extraction expensive;
2. ensure that the extracted CFG is irreducible and that its conversion to reducible form explodes the size of the resulting CFG.

Doing (1) or (2) or both, implies that CFG extraction is quadratic, the data flow analysis is quadratic, or the size of the agent P after conversion to reducible form P' is of the form n^δ, for some $\delta > 1$ (so that even a linear analysis on P' is no longer linear in $|P|$).

Instances of analyses the adversary would have to do are *def-use* and *available expressions* analyses [12, 1]. The first, binds definition statements for a variable x to uses of x within subsequent expressions. Available expression analysis computes, for every program point, the set of previously computed expressions that are still accurate given all variable definitions between the initial computation of the expression and the current program point. On irreducible flow graphs, such analyses require $\Omega(n^2)$ time using the standard iterative algorithm. However, if the flow graph is reducible, they could be performed faster, *e.g.*, in time $\Omega(n \log n)$ using alternative algorithms such as balanced path compression [12]. But again, the required reduction step can increase the size of the CFG drastically.

Figure 3 gives a short sample program and the results of a def-use analysis on it. In the sample program, the adversary must determine the value of `c` returned by the last statement. To do so, it must examine (among many other things) the definitions of variables and their subsequent uses in forming the resultant expression. In the example it is evident that the adversary must uncover `c`'s definition in the `while` loop, which in turn depends on the values of `a` and `b`, which are first initialized before the loop and then repeatedly reassigned in it. Furthermore, using def-use information is often not sufficient as the example also illustrates: arrays are indexed using dynamic values (*e.g.*, `M[a]`) and loops are executed multiple times. Such program constructs complicate the analyses required to even loosely bound, let alone precisely determine, the values and side-effects computed by a TEAS agent. As we argued above, and to summarize, this information cannot be obtained in linear time in an off-line manner using known program analysis techniques.

3.3 Automatic agent generation

An attractive approach to agent generation is to do it automatically on the challenge server, by either precomputing an agent library or by generating agents on-the-fly as needed. Unfortunately, machine generation of programs is insufficiently advanced to create programs anywhere close to as complex and specific as can be created by humans. However, as we describe in this section, it is possible to combine a small (hand-written) program with simple properties with a random, obliviously generated one—we call this process *program blinding.* Program blinding, in conjunction with large sets of suitable agents, can prevent an off-line adversary from adaptively learning the function being blinded, since blinding hides the functionality of the program, and drawing from a large set of programs furhter obfuscates it.

We implemented a virtual register machine (a basic processor instruction set) to test approaches to automatic agent generation. We now turn to the description of an efficient random program generator and program blinding process, followed by an evaluation of results obtained for several program sizes.

Blinding programs with pseudo-random agents. Given a machine language L with its syntax and semantics, or more concretely, its (fixed) set of instructions together with their corresponding actions, we call a *(pseudo-)random program* of size n the result of (pseudo-)randomly generating n valid instructions in L. Note that some of the programs generated this way will not be terminating.

Key to our blinding operation is the notion of *cross-over* taken from genetic algorithms. Given two programs P_1 and P_2, with their respective I/O relations and sets of properties, the cross-over between P_1 and P_2 is a program P_3, obtained by means of some efficient transformation, with a set of properties that contains some of the input programs' properties. In other words, P_3 "inherits" some of the properties possessed by P_1 and/or P_2.

We propose a particular kind of cross-over operation, denoted "$\otimes$," that we call, making an analogy with the notion of "blinding" in cryptography [4], *program blinding.* In a nutshell, given program P, the idea is to combine P with a random program so that the resulting program maintains some of P's properties but its behavior (e.g., its output) is difficult to predict. There would be several ways to produce a juxtaposition of the two programs. In our case, as typically the target program would be much smaller than the random program, we simply define $\otimes$ as the interleaving of each of P's instructions uniformly at random in the random program text. We call $P^* \leftarrow P \otimes P_R$, where P_R is a random program, the *blinded* program.

We now make the "difficult to predict" requirement more precise. We say a program P is (ϵ, n)*-semantically uncertain* if no adversary $A \in \mathcal{A}_{off}$, given $txt(P)$ and an input instance I, can determine $O \leftarrow P(I)$ after n steps of analysis with probability better than ϵ, for all I. In the case of blinded programs, we will set $n = |P^*|$.

Finally, the blinded program's output may or may not depend on the original program's input. We will call those blinded programs where the dependency is preserved *input-sensitive* blinded programs.

Experiments. To validate the idea of using blinding to construct automatically agents we constructed a virtual register machine (VRM) (similar, but simpler than, processor instruction sets). The details of our VRM as described briefly below; a complete description is deferred to the full version of the paper.

We use a simple program that tests a simple property, namely the value of a potentially critical memory location on the *Responder*: `LOADr0([A])`. That is, this program loads the value of memory location `A` into register `r0`. We then blinded this program with a random program of size n drawn from the instructions of our VRM. We then ran the program for at most n^3 instructions where the value at A was taken to be the integer 70 (arbitrarily). If the program did not halt, it was discarded. If it did halt, we recorded its output which was arbitrarily chosen to be the value in register `r1` upon termination. We then reset the program and reran it again with A now holding the integer 50 (again arbitrarily). If the result of the second run differed from the first, we labeled the program as `A`-sensitive. Recall that a sensitive blinded program is one whose output depends on the input of the original program.

Programs labeled as `A`-sensitive by the above procedure are good candidates as agents to test the integrity of location `A`. This is because their result is a function of the value at `A`. Furthermore, they are known to terminate and their number of steps (for any expected value v of `A` can be computed by setting `A` to v and re-running the blinded program).

Table 1 contains results from performing this blinding process 10^5 times for programs of lengths $n = 25$, $n = 50$, and $n = 100$ instructions. The entries in the n, n^2, and

# instr.	n	n^2	n^3	forward jmps	backward jmps
25	687	48	1	2.5	1.9
50	422	59	1	5.3	4.2
100	282	26	1	10.7	9.6

Table 1: *Results for blinding a program P that loads a critical value into a register and returns the final content of this register. P was blinded with random programs of size 25, 50, and 100 instructions. The number of terminating blinded programs sensitive to this value are binned by the number of instructions executed dynamically. Average number of forward and backward jumps per random program are also given.*

n^3 columns are the number of agents out of the 10^5 randomly generated that are A-sensitive. Each column represents an upper bound on the running time in instructions executed. Programs executing more than n^3 instructions were terminated. Generating and testing 10^5 programs took at most a couple of hours (for programs of length 100, less for the shorter lengths). Much of this time is due to interpretation overhead in our VRM and comes mainly from non-terminating programs that were allotted n^3 instruction steps. Nevertheless, random agent generation is fast and can be used either for precomputing a library of agents or for creating them on-the-fly. Note that agents are always tested, so we are always sending tested agents to the client.

For each program length we give the average number of forward and backward branches. Backward branches imply the existence of loops which—as discussed in §2 and illustrated in Figure 1—influence the complexity of program analyses. By biasing the generation procedure, it would be easy to generate agents containing more backward jumps which imply more loops. Some further comments on these results are necessary. First, our sample size of 10^5 is quite small given the size of the search space. This can account for the non-monotonically decreasing counts of the number of quadratic programs with increasing program length. Also, the fact that the experiments produced exactly one program with cubic running time for all three program lengths may be attributable to the sample size. Regarding the jump statistics, note that the average numbers of forward jumps is greater than the average number of backward jumps. This is due to biases in our VRM and in our counting of jumps with an offset of zero as a forward jump. Regardless of such anomalies , the point of the experiments is to demonstrate that program blinding is possible and to give initial statistics for one possible blinding scheme.

The VRM itself consists of a small number of machine-language instructions. For the experiments, we instantiated the VRM with eight registers, of which register R_1 was deemed to hold the agent's result on termination. The `LOADr0()` instruction is special, for it appears only in the program being blinded, $P = \{\texttt{LOADr0}(v)\}$, where v took values 70 and then 50 in these experiments; it is never originally present in a random agent P_R. The blinding process then introduces this instruction into the mix of random instructions that comprise P_R. The result of the blinding, $P^* \leftarrow P \otimes P_R$, is therefore a random sequence of instructions of the VRM with a single `LOADr0(v)` in it. We postulate that by replicating the `LOADr0(v)` instruction more often, one can achieve a higher degree of sensitivity on it which implies more programs matching the sensitivity criteria and faster generation times. Similarly, it should be possible to generate random programs dependent on multiple locations by blinding with programs containing load instructions for the additional locations.

3.4 A construction and security arguments

We now illustrate and analyze a complexity-based TEAS against off-line adversaries, based on the automatic agent generation technique presented above. We focus on one of the analysis-hardening properties discussed in §3.2, that of generating agents with irreducible CFG's.

Let $p^n_{\texttt{irred}}$ denote the probability that a randomly generated agent of size n contains an irreducible CFG (and thus implies a $\Omega(n^2)$ analysis time). Tgen, the TEAS generation algorithm, will be generating instances that include $((1 - p^n_{\texttt{irred}}), n)$-uncertain programs, for suitable n. Depending on the particular generation process, $p^n_{\texttt{irred}}$ can be estimated to various degrees of accuracy.[2] In principle, this probability can be estimated analytically, by for example estimating in the uniform instruction sampling process the probability of loop occurrence with backward jumps occurring into them. Alternatively, if an agent library is precomputed off-line, random agents can be first selected which possess a high number of backward jumps, and then further program analysis (using the best known methods) applied to them. An on-the-fly generation (and testing) process can also be aided by this off-line "knowledge base."

We now explain how Tgen would work to generate an $(\epsilon, \mathcal{A}_{\mathit{off}})$ TEAS instance. First, Tgen is given the target program(s) that is (are) supposed to perform some checks at *Responder*. As mentioned at the beginning of the section, target programs are assumed to be simple programs consisting of a handful of operations. Then, on input CPU rate C at *Responder* and link bandwidth B (and its variance statistics), Tgen chooses n so that the time consumed by a quadratic analysis would dominate the time incurred by a fast-executing (e.g., at most linear) agent by a large factor. How large this factor is would depend on the bandwidth variance statistics. (For example, a factor of three or more assuming a zero probability of half bandwidth—recall the example in §3.2.) This allows Tgen to fix a preliminary patience threshold for all agents (e.g., $\pi = 2$); however, Tgen

[2]It can also be mangled, by for example biasing the sampling procedure.

might perform some additional fine tuning after the next step.

Tgen now proceeds to blind the target program(s) with k terminating random programs of size n such that:

1. every program is input-sensitive; and,
2. the running time of (at least) $k' < k$ agents is at most n, where k' is the minimum integer satisfying $(1 - p^n_{\text{irred}})^{k'} < \epsilon$.

Tgen determines the input sensitivity of the random programs by running them; at this time, besides recording for each blind program P_i^*, $1 \leq i \leq k$, its output o_i, Tgen also computes and records t_i, i.e., the sum of the programs' running time plus their expected transmission time, according to Equation (1). In the case of computation-intensive agents, Tgen might also adjust (lower) their patience threshold, to reflect the fact that computation time for these agents will be the dominating term. Finally, Tgen outputs TEAS instance $\{(P_i^*, o_i, t_i, \pi_i)\}_1^k$ thus constructed, and *Challenger* now submits the agents in the TEAS instance, one by one, to *Responder*.

In order to determine whether *Responder* is corrupted, Tver takes the TEAS instance and the list of pairs $\{(o'_i, t'_i)\}_1^k$, where o'_i is the result provided by agent P_i^* and t'_i is the elapsed time since the agent was sent to *Responder* until the result was received, as measured on *Challenger*'s clock, respectively, and applies the test

$$\texttt{if } \exists P_i^*, 1 \leq i \leq k, \texttt{ s.t. } \quad o_i \neq o'_i \texttt{ OR } \frac{t'_i}{t_i} > \pi_i \texttt{ then} \\ \texttt{output } `\neg\text{OK}'.$$

By construction, the probability that an adversary $A \in \mathcal{A}_{\mathit{off}}$ corrupting *Responder* will pass the test is less than ϵ.

This concludes the section on protection against adversaries who try to discern program properties by performing off-line analysis. As we shall see in the next section, an adversary that emulates or interprets an agent may be able to determine properties with only moderate overhead. We now show how to structure agents to avoid such on-line adversarial attacks.

4. INTEGRITY PROTECTION FROM ON-LINE ADVERSARIES

To illustrate, consider this example. An adversary corrupting *Responder* is trying to protect a memory location ℓ. As a concrete example, ℓ may contain configuration data that has been changed to modify the characteristics of a set-top box. When *Responder* now receives a TEAS agent P, the adversary applies analyses to it before running it to determine if the memory locations accessed by P contain ℓ. If the adversary is able to determine that P accesses ℓ, it could restore ℓ to the proper value, and cache the improper malicious value at a location ℓ' that it has determined not to be accessed by P. When P completes it returns a correct value to the *Challenger* (since ℓ contained the proper value), but then the adversary can reinstall its improper configuration into ℓ.

In fact, it would be easy to articulate such an *interpreter attack* by an on-line adversary, assuming a memory map as depicted in Figure 2. Essentially, and assuming the existence of some unused area of memory (free space, or unused space in the data area) at *Responder*'s memory, the adversary can control the execution of the agent in such a way that loads and stores from/to protected regions, as well as executions of pieces of code in such regions are always avoided. Note that the overhead for interpreting the agent would be constant (say, five to ten instructions per simulated instruction).

We now present an on-line defense construction and corresponding security arguments. At a high level, our strategy for defeating interpreter attacks will be to generate a TEAS instance that will either force the adversary to relinquish control of *Responder* if the instance is executed completely, or risk detection by again not being able to provide results in a timely manner. In more detail, we assume that *Challenger* has a detailed knowledge of the code area; recall also that we do not allow the adversary change any of the client's hardware, in particular its memory size (M). Tgen now generates TEAS instances having the basic structure of agents $\{P_1, P_2, P_3\}$, where:

- P_1 is an agent carrying code (or arguments) for a pseudo-random permutation of *Responder*'s memory. P_1 returns a `'ready'` message upon completion to *Challenger*. Figure 4 shows the fragment of such an agent. Note that in an actual implementation this permutation code would insure that the agent itself, and any support routines required by it such as network drivers for talking to the *Responder*, are not permuted. This is possible since we can assume the agent knows where it and its support routines are located in memory.
- P_2 carries a sequence of (now random) locations of memory and program control locations (e.g., the program stack) whose contents are to be returned to *Challenger*. (P_2 could itself consist of several agents.)
- P_3 is an agent carrying code (or arguments) for the inverse pseudo-random permutation that P_1 performed. I.e., P_3 restores the state of *Responder*.

First assume that P_1, the agent carrying the permutation code, gets executed at *Responder*. Then, necessarily, *Responder* will be under control of the agent. P_2, the agent carrying the queries, will now be able to inspect *Responder*'s configuration. This is possible since *Challenger*, knowing the seed of the pseudo-random permutation, knows how the memory locations get mapped because of it. By the same token, the execution of the queries should be very fast. Additionally—and importantly— P_2 will also be able to inspect the program call stack, which should point to areas of valid code, whose contents can also be verified. Should Tver detect a time violation in the execution of P_1 or P_2, or

```
/* agent fragment */

a2 = seed;
for (i = 0; i < M; i++) {
      a1 = random(a2);
      a2 = random(a1);
      t = M[a1];
      M[a1] = M[a2];
      M[a2] = t;
}
```

Figure 4: *Random permutation of* all *of Responder's memory. An actual implementation would ensure that the agent and its support routines are not permuted during this operation.*

receive an incorrect result from P_2, it outputs '$\neg$OK'.

If, on the other hand, P_1 does not get executed at *Responder*, then the probability of correctly responding to a P_2 query is, roughly, $\frac{1}{M}$. Alternatively, the adversary could also retain control of *Responder* by computing the memory mapping *without* performing the actual permutation of memory locations by constructing a table containing a mapping from a location to its permuted location. This would allow an interpreter attack to decipher location queries and thereby provide correct values in response to a memory query in P_2. This is possible since the adversary would know the seed that P_1 carries. However, the memory requirements for such a table would be $O(M)$ and the adversary would face the risk of being disabled also in this case. For example, in the limit *Challenger* might permute all locations which would force the adversary to create and maintain a table larger than its available memory. Again, partial construction of such table would have a low probability of success in answering P_2's queries.

5. APPLICATIONS OF TEAS

TEAS are general and can be adapted to many applications for which system integrity is to be tested. To convey this, we describe three systems in which it could be used:

- terminal devices such as mobile phones or set-top boxes,
- computing "appliances" such as wireless basestations, and
- sensor networks.

Following a discussion of these, we address some practical issues regarding TEAS implementations.

Terminal devices. For the first TEAS application, consider a network of set-top boxes (or mobile phones) where the provider owns both the network and the terminal devices (*e.g.*, the cable network and cable modems, or a satellite television system). It is of extreme interest to the provider to ensure that the terminal devices are not misused or that rogue devices do not receive service. Malicious users routinely "hack" cable-modems, for example, to up the bandwidth delivered to them; similarly, people reprogram their satellite receivers to get additional unauthorized services. Using TEAS one can help protect against such misuse by periodically sending an agent to inspect the state of active cable modems (or satellite receivers). Since the provider typically owns all of the infrastructure, device and timing parameters are fully known. Furthermore, transmission times from the network's edge routers to a cable modem can be quite deterministic. Modifications of the terminal hardware is much more difficult (for both the provider and a malicious hacker) than reprogramming its software, making such a system an ideal candidate for integrity protection via TEAS.

Computing appliances. Define a *computing appliance* as a tightly coupled collection of co-located computing devices. An example of this is a wireless basestation that contains a controller board (computer) and a number of channel cards (also containing microcontrollers). The network connecting these components is a typically a fast backplane with highly deterministic transit times. Often basestation channel cards can operate at various capacity levels depending on the license purchased with the card. The operating capacity, if only a software configuration, is changeable by unscrupulous operators. TEAS can aid in protecting such a system by having the controller card serve as the *Challenger* and all channel cards as *Responders*. As in the above example, agents would query critical configuration information.

Sensor networks. Our third example, that of sensor networks, is similar to the first since it too might span a geographically large area with associated network latencies. Different however is the threat level. In a military or critical mission application it is of utmost importance that data gathered by the sensors be true. TEAS can contribute here because of the flexibility of the challenges (full-fledged programs). If for instance one surmises a sensor has been coopted and is being operated by a malicious entity, custom challenges can be designed to probe our unauthorized modification of the software. Furthermore, agents could in some cases even be used to check for hardware tampering if the hardware modifications alter the operation of *some* software.

Practical issues. Here we address practical issues facing TEAS implementations. They concern two aspects of TEAS: first, setting the parameters required for agent selection and generation (§2 and §3.3); and second, implementation details of the client *Responders*.

Parameters of the agent model require fairly precise specification of the computation rates on the *Responders*as well as of the communication rates between the *Challenger* and its *Responders*. Since we assume that both the *Challenger* and the *Responder* are operated by the same entity, it is fairly easy to establish device characteristics empirically in the lab. For instance, one can precisely measure the time required per instruction on the *Responder* processor(s). Similarly, one can measure the bandwidth and network latencies by sending large random[3] streams and measuring actual throughput and latency. Variance statistics can then be computed from the empirical measurements to give acceptable operating ranges.

The TEAS implementation of agents must be compatible with the operating system and environment of the client *Responder*'s on which it will run. In particular, system issues such as *process scheduling* and *interrupts* impact the design of a TEAS system. However, since a TEAS agent is small relative to modern software (tens of thousands of bytes is a large agent) and its execution is swift, it will not interfere with all but the most time-critical applications on the *Responder*. For example, in many applications it suffices to suspend the current application's processes and to disable interrupts immediately upon receipt of an agent. Upon receipt, the agent authenticated and loaded into the memory it specifies, is run, and its result returned to the *Challenger*. The *Responder*'s application is now resumed.

6. RELATED WORK

This paper's introduction already introduced the primary previous work related to TEAS. Here we further elaborate on the prior Genuinity and SWATT systems and also discuss work related to agent security in mobile systems.

Recently, Kennelly and Jamieson [11] also suggested having a client compute a challenge problem in order to verify that it is running the correct version of uncorrupted software (specifically, a kernel). In their *Genuinity* system, the challenge problem is a checksum of the contents of pseudorandom (virtual) memory addresses and other processor state; the verification code resides at the client, and the sequence of pseudorandom addresses is sent by the secure host to the client in the form of a linear-feedback shift register. The client constructs the checksum by adding the one-byte values at those virtual addresses. Between additions, the code also incorporates other, hardware-specific (Intel x86 architecture) values into the checksum, such as the mapping of a particular instruction or Data TLB entry, if it exists; instruction or data cache tags; values of performance counters which measure the number of (branch) instructions encountered; etc. The trusted host also computes the checksum, and if the client being tested calculates the same checksum as the trusted host and returns it "quickly enough," the client software is considered genuine, since it is assumed that there would be no fast simulation.

In their refutation of Genuinity [15], Shankar, Chew and Tygar present a "fast enough" simulation (with running time below the 35% slowdown allowed by Genuinity), by mounting what they call a *substitution attack* on the system, which is similar to what we call the *interpreter attack* by on-line adversaries: Imposter code that sits in the client's checksum page available memory that makes sure that correct values are substituted for malicious ones in the checksum. TEAS differs from Genuinity in several ways. First is the general formulation and variety (and unpredictability) of challenges that TEAS can draw from. Second is the technique to prevent the substitution (interpreter) attack, which makes it difficult for the imposter code (on-line adversary) to perform in a timely manner without being disabled. In fact, this makes the TEAS techniques for the off-line scenario yet more relevant, since should an adversary corrupting the client try to return the correct responses without running the challenges, in order to avoid being disabled, it would face solving the difficult program analysis problems that TEAS can pose.

[3]Random, to avoid compression mechanisms in the network hardware.

The other system directly relevant to TEAS is Seshadri *et al.*'s SWATT system [16]. SWATT applies the so-called *coupon collector problem* to probabilistically ensure that a random traversal of $O(n \log n)$ memory locations (in a memory of size n) will almost always visit every location at least once. In SWATT as implemented, the verification code resides on the machine being verified. This code is supplied a seed to a PRG by the challenger. The PRG then governs the memory traversal order. The large amount of the work required by the verification code makes it practical only on small memory sizes. Furthermore, SWATT requires the challenger to know the entire content of memory exactly, which is probably not the case for non-trivial applications. Two other key points are that (1) the time required by the verification code is crucial to SWATT's success and relatedly that (2) the verification code is "optimal" in time (to ensure the accuracy of the its timing by the challenger). This implies that SWATT is applicable only to very deterministic processors (no caches, for instance). Furthermore, it requires a tight coupling between challenger and responder since network latencies cannot be incurred in SWATT. It is not specified how the device being challenged is even made aware that a challenge is present on an input port since the device could be busy with a lengthy computation. The SWATT authors do not address how interrupting the device can severely affect their tight timing constraints necessary for their scheme to work in practice.

We believe that ideas in TEAS, combined with the advances in the Genuinity and SWATT systems, can make software-based verification more practical as well as more sound. In particular, TEAS can tolerate some latencies between *Challenger* and *Responder* due to its probabilistic nature. Furthermore, it can cope with the commonplace scenario where the state of the *Responder* is only partially known at any point in time. Finally, the general nature of TEAS' agents arguably makes an adversaries task of maintaining a malicious presence much more difficult, if not impossible.

There is a large body of work dedicated to security issues that arise in mobile code systems (see, e.g., [19] and references therein). With a few exceptions (e.g., [14]), most prior work focuses on security mechanisms and policies that would determine whether the requests issued by the mobile code should be allowed or denied by the (remote) host executing the code. Work that considers the case of malicious hosts includes the work by Esparza *et al.* [9] where the execution time of an agent is also used to detect possible malicious behavior by the host; the purpose of that work, however, is to protect the agent against abuses from a malicious host. Specifically, each host on the agent's itinerary is required to save the agent's arrival and departing time. When the agent returns to the host of origin, a set of checks verifies that no host spent more time than expected executing the agent Naturally, this approach works only if measures can be taken to prevent malicious hosts from misrepresenting an agent's incoming and departing times. In contrast, our use of time checks is founded on the design of problems that an adversary would, in all likehood, not be able to solve in a timely manner.

7. CONCLUSIONS AND FUTURE WORK

This paper presented TEAS, a general approach toward improving the security of systems that, for whatever reason, cannot or are not protected by hardware means. TEAS consists of a trusted challenger that issues challenges to a perhaps corrupted client responder. New is that the challenges can be *arbitrary programs* and can therefore perform intricate tasks on the client. Furthermore, challenges are timed. Using results from programming languages and computing theory, we show that agents may be constructed in ways to make it very difficult for an adversary to discern their behavior within acceptable time bounds. Finally, we introduced techniques toward automatic derivation of agents with such properties, which we call program blinding. As such, TEAS can augment existing protection mechanisms.

TEAS are general and can be adapted to many applications for which system integrity is to be tested, such as (1) terminal devices such as mobile phones or set-top boxes, (2) computing "appliances" such as wireless basestations, and (3) sensor networks. However, future work is necessary to incorporate features of high performance processors, such as caches and virtual memory, into the basic model of our responder's processor (Section 2). Such features would impact reliance on deterministic timing assumptions.

Many extensions and refinements to TEAS are possible. The construction of agents that are difficult for adversaries to analyze efficiently is of interest and valuable. Such study can improve effectiveness of the program blinding procedure. In particular, future work can be directed toward characterizing the "reducibility" of their flow graphs (*cf.* §3).

In conclusion, we have described a flexible software-only system that can be used to check the integrity of a computing system within a trusted domain. We gave methods for creating security agents and argued that an adversary's analysis of these is necessarily difficult. We believe the TEAS can become a valuable component in the tools available to implementors of secure systems.

8. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Shai, S. Vadhan, and K. Yang. On the (Im)possibility of Obfuscating Programs. In *Proc. CRYPTO '02*, LNCS(2139), pp. 1-18, Springer-Verlag, August 2001.

[3] D. Boneh and M. Naor. Timed commitments (extended abstract). In *Proc. CRYPTO '00*, LNCS(1880), pp. 236–254, Springer-Verlag, August 2000.

[4] D. Chaum. Blind signatures for untraceable payments. In *Proc. CRYPTO '82*, pp. 199–203. Plenum Press, New York and London, 1983, August 1982.

[5] C. Collberg and C. Thomborson. Watermarking, Tamper-Proofing, and Obfuscation–Tools for Software Protection. IEEE Trans. on Software Engineering, Vol. 28, No. 8, Aug. 2002.

[6] C. Collberg, C. Thomborson and D. Low. A Taxonomy of Obfuscating Transformations. Technical Report #148, Dept. of Computer Science, University of Auckland, 1997.

[7] K. D. Cooper, T. J. Harvey and T. Waterman. Building a Control-Flow Graph from Scheduled Assembly Code. Technical Report #TR02-399, Rice University, June 2002.

[8] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *Proc. CRYPTO '92*, LNCS(740), pp. 139–147, Springer-Verlag, August 1992.

[9] O. Esparza, M. Soriano, J. Muñoz and J. Forné. Limiting the execution time in a host: a way of protecting mobile agents. In *IEEE Sarnoff Symposium on "Advances in Wired and Wireless Communications,"* 2003.

[10] J. E. Hopcroft and J. D. Ullman. *Int. to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[11] R. Kennell and L. Jamieson. Establishing the genuinity of remote computer systems. In *12th USENIX Security Symposium*, pp. 295-310, 2003.

[12] S. S. Muchnick and N. D. Jones. *Program Flow Analysis: Theory and Applications* Prentice-Hall, 1981.

[13] R. Rivest, A. Shamir, and D. Wagner. Time-lock puzzles and timed-release crypto. *MIT/LCS/TR-684*, 1996.

[14] T. Sanders and C. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile Agents and Security*, LNCS(1419), Springer-Verlag, 1998.

[15] U. Shankar, M. Chew and J.D. Tygar. Side effects are not sufficient to authenticate software. In In *13th USNIX Security Symposium*, pp. 89-101, 2004.

[16] A. Seshadri, A. Perrig, L. van Doorn and P. Khosla, SWATT: SoftWare-Based ATTestation for Embedded Devices. In *IEEE Symp. on Security and Privacy*, 2004.

[17] S. Smith, R. Perez, S. Weingard, and V. Austel. Validating a high-performance, programmable secure coprocessor. In *22nd National Information Systems Security Conference.* October 1999.

[18] Trusted Computing Group (TCG). `https://www.trustedcomputinggroup.org/`, 2003.

[19] D. Wallach. A New Approach to Mobile Code Security. Ph.D. Dissertation, Princeton University, January 1999.

[20] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, 1989.