Robustness Analysis for Secure Software Design

Eunsuk Kang
Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
eskang@cmu.edu

ABSTRACT

A common type of security analysis involves checking whether a system is capable of establishing a set of security requirements under a particular threat model. Building an accurate threat model, however, is a challenging task due to the uncertain and evolving nature of a malicious environment in which the system is deployed. In this paper, as a complementary analysis, we propose a systematic approach for evaluating the design of a system with respect to its *robustness* against an adversarial environment; i.e., the degree of assumptions about attacker capabilities under which the system is capable of maintaining its security requirements. We argue that robustness is an important property that should be considered as part of any secure development process. In this paper, we propose a formal definition of robustness, and describe a technique for automatically evaluating the robustness of a system. We demonstrate potential applications of the robustness concept using an example involving the OAuth authentication protocol.

CCS CONCEPTS

• **Software and its engineering** → **Extra-functional properties**; **Software organization and properties**.

KEYWORDS

Security, design, robustness, assumptions, modeling

ACM Reference Format:
Eunsuk Kang. 2020. Robustness Analysis for Secure Software Design. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Security from Design to Deployment (SEAD '20), November 9, 2020, Virtual, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3416507.3423191

1 INTRODUCTION

A common type of security analysis centers around the following question: Given a system (S), a desired security requirement (R) and a *threat model* (T) that describes the capabilities of an attacker, does the system satisfy the requirement under the presence of such an attacker (i.e., $S \parallel T \models R$)?

One of the most challenging steps in this process is building an accurate threat model. Modern software systems are deployed in a highly dynamic, heterogenous environment (such as the Internet), and it is difficult, if not impossible, to identify all relevant ways in which a malicious actor may attempt to compromise the system. To make the matter worse, as the environment evolves over time, the threat model that the designer originally had in mind may become invalid, undermining the security of the system. For example, OAuth, a popular authorization protocol, has been shown to be vulnerable to new attacks when it became re-purposed for deployment in mobile applications (beyond the original target of websites) [9].

In this paper, we propose a complementary type of analysis: Given a system and a security requirement, what are the *strongest attackers* against which the system is capable of satisfying the requirement? In particular, we use the term *robustness* (Δ) to denote the capabilities of such attackers. Robustness captures *the extent to which the system can tolerate the behaviors of malicious actors while providing a desired level of security*; i.e., the system is secure against any attackers that are no more powerful than those in Δ. On the other hand, under the presence of an attacker that has more capabilities than captured by Δ, the system may no longer be able to guarantee the same level of security.

Intuitively, robustness encodes a set of *assumptions* that the system makes about potential adversaries in its environment. For example, such an assumption may describe which of the system interfaces the attacker is able to manipulate, or whether the attacker has knowledge of a potentially sensitive piece of information about users. These assumptions, however, often remain implicit during a development process, and when violated, may undermine the security of the system [4, 38]; our goal is to provide a systematic approach for explicating and reasoning about assumptions.

We further argue that robustness itself can be considered a property of software design, and that it enables a new set of security analysis tasks to be performed as part of a secure deployment cycle:

- Will my system be secure if deployed in the current environment? (i.e., is the current threat model T weaker than Δ?)
- How much uncertainty or change in the threat model can my system tolerate? (i.e., what additional attacker capabilities in Δ beyond T can S tolerate?)
- Given a pair of alternative designs (S_1 and S_2), is one more robust than the other? (i.e., does Δ_1 describe stronger attackers than Δ_2 does, or vice-versa?)

In this paper, to systematize and enable tool support for these analysis tasks, we first propose a formal definition of robustness. Then, building on this foundation, we describe a technique for automatically computing robustness from a formal description of a system and its security requirement. We demonstrate our approach through an example involving the design of the popular OAuth authorization protocol [25].

2 EXAMPLE

As a motivating example, consider OAuth [25], a popular third-party web authorization protocol. It is designed to allow an application (called a *client* in the OAuth terminology) to access a resource owned by a user (*user-agent*) on another application (*authorization server*) without needing the user's credentials. Although the protocol was designed for third-party *authorization*, its most common usage is for *authentication*, where the user is given an option of authenticating themselves with a client by using existing credentials from another organization (e.g., Google, Facebook), instead of creating a new account with the client.

A high-level overview of the protocol, based on the official specification for OAuth 2.0 [25], is shown in Figure 1. A typical OAuth session begins with the user initiating the protocol with the client (Step 1). The user must then prove her identity to the server through authentication (Step 2), after which the server provides a unique *authorization code* to be forwarded to the client (Steps 3-4). Having obtained this code from the user, the client then presents it to the server in order to obtain a corresponding *access token* (Steps 5-6), which can subsequently be used to access the user's resources.

One of the desirable security requirements of OAuth is called the *session integrity* property [16]; i.e.,

> *When the client receives an access token at the end of the protocol, the token must represent the user who initiated the same protocol session.*

This requirement may be violated, for example, when a malicious actor (called Eve) is able to deceive the client into associating an access token intended for a victim user (Alice) with Eve's account on the client; consequently, Eve may be able to access Alice's resources on without needing the latter's credentials.

Suppose that we are given the task of analyzing the protocol to check whether it satisfies the desired security requirement (in this case, session integrity). A typical next step would involve coming up with a threat model that describes the capabilities of an attacker. In the context of OAuth, the attacker may carry out a variety of malicious actions in order to compromise the requirement. For example, the attacker may impersonate a user agent, interacting with the client to initiate a protocol session and obtain an authorization code from the server. Alternatively, the attacker may also pose as the client or the server, in order to manipulate the user agent into forwarding an authorization code to an unintended location and undermine session integrity [32, 36].

Coming up with an accurate and comprehensive threat model, however, is a time-consuming and challenging task. The official threat model for OAuth 2.0 [18], for example, lists over 40 potential threats (i.e., attacker actions); devising such a list requires a significant amount of security expertise, beyond that of an average software developer. In addition, not all threats are equal, and only some subset of these may be relevant to a particular security requirement being established.

A different kind of security analysis is to ask the following: Under what types of attackers can the protocol ensure session integrity?

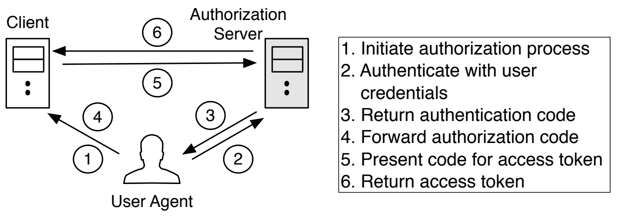
Client
Authorization Server
User Agent
1
2
3
4
5
6
1. Initiate authorization process
2. Authenticate with user credentials
3. Return authentication code
4. Forward authorization code
5. Present code for access token
6. Return access token

Figure 1: Overview of OAuth 2.0

This information (which we refer to by symbol Δ) may be useful to the developer in a number of ways. First, by comparing Δ to the list of known threats on a target deployment environment (e.g., web browser), the developer can decide whether it is secure to deploy the protocol as it is, or whether additional security measures are needed to address those threats that do not belong to Δ. As we will later demonstrate in Section 4.2, the developer may also compare Δ's of alternative protocols to determine which of them may be more secure to use.

Our goal is to systemize and enable tool support for this type of analysis. In the rest of the paper, we report on our preliminary work towards this goal. In particular, we propose (1) a formal definition of robustness for security, and (2) an approach for automatically computing robustness from a formal specification of a system and a desired security requirement.

3 ROBUSTNESS IN SECURITY

In this section, we first describe our approach to modeling the behavior of a system and the capabilities of attackers. Then, we propose a formal definition of robustness as the capabilities of the most powerful attackers under which system can still satisfy a desired security requirement.

3.1 Modeling Approach

Processes. In our approach, a system consists of a set of *processes* that interact with each other by performing various types of *events*. Let P denote the set of processes, E the set of events, and T the set of *traces*, each of which is a finite sequence of events. Each process p is associated with an *alphabet* (denoted $\alpha(p) \subseteq E$), which describes the set of events that it may perform. Trace t is considered a trace of process p if t describes one possible sequence of events that can be observed from p. Then, the overall behavior of p, denoted $beh(t) \subseteq T$, is defined as the set of all traces of p.

To model interactions between a pair of processes, p and q, we use the *parallel composition* operator ($p \| q$) similar to that of the CSP process algebra [23]. In this composition, the processes interact by synchronizing on events that are common to their alphabets; the overall behavior of the composed process is defined as follows:

$$\begin{aligned} beh(p \| q) = \{t \in T \mid\ & t \in (\alpha(p) \cup \alpha(q))^* \wedge \\ & (t \upharpoonright \alpha(p)) \in beh(p) \wedge (t \upharpoonright \alpha(q)) \in beh(q)\} \end{aligned}$$

where $(t \upharpoonright X)$ is the projection of t onto the event set X.

Types. Each process p is associated with one or more *types*, which are used to restrict the kinds of behaviors that the process is allowed to engage in. The *signature* of type τ, denoted $\sigma(\tau) \subseteq E$, is the set of events that can be performed by a process that has τ as one of its types. More formally, the following constraint is imposed on our models of systems:

$$\forall p \in P, e \in \alpha(p) \cdot \exists \tau \in types(p) \cdot e \in \sigma(\tau)$$

One may find that allowing a process to be associated with multiple types to be unusual. As we will see later, this allows a malicious process to take on multiple roles (e.g., in OAuth, impersonating as a client and an authorization server) with the goal of compromising a system requirement.

Knowledge. Each process possesses knowledge about various types of data that are used in the system (e.g., keys, messages, nonces). The knowledge of a process may grow over time as it interacts with other processes through events. More precisely, let *data value* $d \in D$ be an atomic entity that can be transmitted from one process to another as one of the parameters of an event ($params(e) \subseteq D$). Furthermore, each process is assigned a set of initial knowledge that it possesses at the beginning of the system execution ($init(p) \subseteq D$).

Then, the *knowledge* of the process, denoted $knows(p, t)$, is defined as the set of values that are accessible to p after executing trace t:

$$knows(p, t) = \{d \in D \mid d \in init(p) \vee \\ \exists e \in E \cdot e \in events(t \upharpoonright \alpha(p)) \wedge d \in params(e)\}$$

Informally, this states that process p can access a piece of data d either by possessing it as part of its initial knowledge or engaging in an event that carries the data as a parameter.

Example. A snippet of a model of the OAuth protocol is shown below[1]:

$$P = \{\mathsf{Alice}, \mathsf{Eve}, \mathsf{MyApp}, \mathsf{Google}\}$$
$$types(\mathsf{Alice}) = \{\mathsf{UserAgent}\} \; init(\mathsf{Alice}) = \{\mathsf{aliceUID}, \mathsf{alicePwd}\}$$
$$types(\mathsf{MyApp}) = \{\mathsf{Client}\} \quad types(\mathsf{Google}) = \{\mathsf{AuthServer}\}$$
$$init(\mathsf{Google}) = \{\mathsf{aliceCode}, \mathsf{eveCode}, \mathsf{aliceUID}, \mathsf{alicePwd}, \ldots\}$$
$$\sigma(\mathsf{Client}) = \{\mathsf{InitOAuth}, \mathsf{ReturnCode}, \mathsf{GetAccessToken}\}$$
$$\sigma(\mathsf{AuthServer}) = \{\mathsf{Authenticate}, \mathsf{GetAccessToken}\}$$
$$\sigma(\mathsf{UserAgent}) = \{\mathsf{InitOAuth}, \mathsf{Authenticate}, \mathsf{ReturnCode}\}$$

The model consists of four processes, which interact with each other through events that are common to their alphabets. For example, Alice and MyApp (which are of type UserAgent and Client, respectively) can interact with each other by performing two types of events: InitOAuth and ReturnCode, corresponding to Steps 1 and 3, respectively, of the protocol workflow shown in Figure 1.

3.2 Attacker Capabilities

In our approach, the capabilities of an attacker are expressed in terms of two concepts: (1) the amount of interactions that it is able to engage in with other processes, and (2) knowledge about the system that it may leverage for its attacks. For instance, one type of attacker on OAuth may be capable of impersonating a client and interacting with a victim user agent (e.g., Alice) through InitOAuth and ReturnCode events. The same attacker may initially have access to some of the information about Alice (e.g., aliceUID, but not her password alicePwd), and grow its knowledge over time through the interaction with Alice (e.g., obtaining an authorization code when Alice invokes ReturnCode on the attacker impersonating the client).

Formally, in our modeling approach, some of the processes in the system are designated to be *untrusted* (i.e., $untrusted \subseteq P$). For example, in the OAuth example, we designate a process named Eve to be one that is potentially under the control of an attacker:

$$untrusted = \{\mathsf{Eve}\}$$

The capabilities of the attacker are then represented by the types and initial knowledge of the untrusted processes. For instance, in our OAuth example, suppose that we are given the following:

$$types(\mathsf{Eve}_1) = \{\mathsf{UserAgent}\} \quad init(\mathsf{Eve}_1) = \{\mathsf{eveUID}, \mathsf{evePwd}, \mathsf{aliceUID}\}$$

The process Eve_1 represents an attacker who is capable of acting like an user agent and interacts with Client and AuthServer processes through events InitOAuth, Authenticate, and ReturnCode. In addition to its own ID and password, Eve_1 is also given the knowledge of Alice's user ID to begin with.

A different type of attacker may be specified as follows:

$$types(\mathsf{Eve}_2) = \{\mathsf{UserAgent}, \mathsf{Client}, \mathsf{AuthServer}\}$$
$$init(\mathsf{Eve}_2) = \{\mathsf{eveUID}, \mathsf{evePwd}, \mathsf{aliceUID}\}$$

Under this threat model, Eve_2 is capable of impersonating any of the participants involved in the OAuth protocol. Arguably, Eve_2 is a more powerful attacker than Eve_1, since Eve_2 can engage in any of the system events (i.e., $\alpha(\mathsf{Eve}_2) = \sigma(\mathsf{UserAgent}) \cup \sigma(\mathsf{AuthServer}) \cup \sigma(\mathsf{Client})$) and has more influence on the protocol outcome.

3.3 Robustness Definition

Before defining robustness, we first define what it means for a process to be more or less powerful than another.

Definition 3.1. Given a pair of processes, $p, q \in P$, $p \leq q$ if and only if $types(p) \subseteq types(q)$ and $init(p) \subseteq init(q)$. Process q is also said to be *at least as powerful* as p.

Informally, process q is at least as powerful as p if q can behave like any of the processes that p can, and q has as much initial knowledge as p. Note that ordering $\leq$ forms a partial order over the set of processes. A pair of processes p and q may be *incomparable*, in that p is not more powerful than q, or vice-versa. For instance, suppose that

$$types(p) = \{\mathsf{UserAgent}\} \quad types(q) = \{\mathsf{Client}\}$$

Process p and q are each capable to impersonating different components and thus are not considered more powerful than each other.

Then, our notion of robustness for security is defined as follows:

Definition 3.2. Given process S and security requirement R, the *robustness* of S with respect to R, denoted $\Delta(S, R)$, is a set of processes ($T \subseteq P$) such that:

(1) For every $p \in T$, $S \parallel p \models R$,
(2) For every $p' \in P$ such that $S \parallel p' \models R$, there exists some $p \in T$ such that $p' \leq p$, and
(3) For every $p, q \in T$ such that $p \neq q$, $\neg(p \leq q \vee q \leq p)$.

[1]Here, we focus on the structural aspect (i.e., types) of the protocol and omit the behavioral specification of the processes, since the latter is not one of our contributions. A paper detailing a trace-based model of OAuth (which also helped inform our model) can be found in [16].

Formally, statements (1), (2) and (3) together ensure that Δ is the set of all *maximal* elements of the partially ordered set of processes that, when composed with S, satisfy R.

Informally, the robustness of system S represents the set of the most powerful attackers under which the system is capable of establishing security requirement R. Conceptually, Δ can be thought of as the "upper bound" on the malicious behaviors in the environment that the system can tolerate. More powerful the attackers in Δ are, more desirable it is, since it means that the system can be securely deployed in a wider range of environments.

In the OAuth example, overall system process S is constructed as the composition of trusted processes; i.e., $S = \mathsf{Alice} \parallel \mathsf{MyApp} \parallel \mathsf{Google}$. Then, its robustness $\Delta(S, R)$ (where R is a specification of the session integrity requirement) describes the most powerful version(s) of Eve under which S can still ensure R.

4 ANALYSIS

4.1 Computing Robustness

We propose an approach for automatically computing robustness given a model of the system model and a desired security requirement. In particular, we assume that the system model and requirement are specified in a declarative, constraint-based specification language like Alloy [26]. Due to limited space, we will omit the details of how the style of security modeling described in Section 3.1 can be specified in Alloy[2].

Constraint-based specification. A specification $\mathcal{S}$ is a conjunction of first-order logic constraints (i.e., $\mathcal{S} = C_1 \wedge C_2 \wedge ... \wedge C_n$) over some set of variables $v_1, v_2, ..., v_k \in V$. The specification $\mathcal{S}$ is *satisfiable* if and only if there exists at least one *instance* that involves an assignment of values to all of the variables; $\mathcal{S}$ is said to be *unsatisfiable* if no such instance exists, indicating that the set of constraints are logically contradictory. An *unsatisfiable core UC* is some subset of constraints (i.e., $UC \subseteq \{C_1, C_2, ..., C_n\}$) such that UC is unsatisfiable [33]. An unsatisfiable core is said to be *minimal* if removing any of the constraints from UC renders it satisfiable. In general, a specification may contain *multiple* minimal unsatisfiable cores.

Suppose that we are given a constraint-based encoding (C_S, C_T, C_R) of a process-based model from Section 3.1, where C_S specifies the behavior of trusted processes (e.g., Alice, MyApp, and Google in the OAuth example), C_T the untrusted processes (e.g., Eve), and C_R the security requirement to be checked. Then, analyzing whether the system can establish requirement R under attacker T can be formulated as the problem of checking the unsatisfiability of the following specification:

$$\mathcal{S}(S, T, R) \equiv C_S \wedge C_T \wedge \neg C_R$$

A satisfying instance to $\mathcal{S}$ would correspond to a possible system trace where the requirement is violated by the actions of the attacker. If, on the other hand, $\mathcal{S}$ is unsatisfiable, it means no such trace exists and thus, the system can establish R under attacker T.

Approach. The high-level idea behind our approach to computing robustness is to use a minimal unsatisfiable core to identify constraints over attacker capabilities that are needed to establish the given requirement. Let T_w be a process that describes the *weakest* attacker; i.e.,

$$types(T_w) = \emptyset \quad init(T_w) = \emptyset$$

One way to specify such an attacker using constraints is as $C_{T_w} \equiv C_{types} \wedge C_{init}$, where:

$$C_{types} \equiv \bigwedge(\tau \notin types(T)) \text{ for each type } \tau \in \mathcal{T}$$
$$C_{init} \equiv \bigwedge(d \notin init(T)) \text{ for each data value } d \in D$$

Intuitively, each of these constraints encodes a restriction about the attacker capabilities—namely, that the attacker is not capable of acting like a process of type τ, or that it does not initially have access to a particular piece of data d.

For instance, the weakest attacker for the OAuth model can be specified as follows:

$$\begin{aligned} C_{T_w} \equiv\ & \mathsf{UserAgent} \notin types(\mathsf{Eve}) \wedge \mathsf{AuthServer} \notin types(\mathsf{Eve}) \wedge \\ & \mathsf{Client} \notin types(\mathsf{Eve}) \wedge \\ & \mathsf{aliceUID} \notin init(\mathsf{Eve}) \wedge \mathsf{alicePwd} \notin init(\mathsf{Eve}) \wedge \\ & \mathsf{aliceCode} \notin init(\mathsf{Eve}) \wedge \mathsf{eveCode} \notin init(\mathsf{Eve}) \wedge ... \end{aligned}$$

Under the weakest attacker, the system should satisfy the given requirement; i.e., $\mathcal{S}(S, T_w, R) \equiv C_S \wedge C_{T_w} \wedge \neg C_R$ is unsatisfiable. A minimal unsatisfiable core (MUC) extracted from this specification contains some subset of constraints from C_{T_w}; let us call this subset MUC_T. By definition of a minimal core, if any constraint is removed from MUC_T, the resulting specification will be satisfiable (i.e., there will be a violation of the requirement). Thus, MUC_T is a minimal set of constraints on the attacker capabilities needed for the system to establish its requirement.

THEOREM 4.1. *Given system S and requirement R, let C_S and C_R represent a constraint-based specification of S and R, respectively; let C_{T_w} be a specification of the weakest attacker. Furthermore, let MUC be a minimal unsatisfiable core for specification $\mathcal{S}(S, T_w, R)$, and MUC_T be the restriction of MUC to those constraints in C_{T_w}. Then, the attacker capabilities represented by MUC_T is equivalent to one of the attackers in $\Delta(S, R)$.*

Recall that specification $\mathcal{S}$ may contain multiple MUCs in general. In particular, each of these MUCs describes *one* of the maximal attackers in $\Delta(S, R)$ that are incomparable with each other with respect to the ordering relation $\preceq$ defined in Section 3.3. Thus, robustness $\Delta(S, R)$ can be computed by finding the set of all MUCs to specification $\mathcal{S}$.

THEOREM 4.2. *Let $\mathcal{M} = \{MUC_1, ..., MUC_k\}$ be the set of all minimal unsatisfiable cores for specification $\mathcal{S}(S, T_w, R)$, and $\mathcal{M}_T = \{MUC_{T_1}, ..., MUC_{T_k}\}$ be their restrictions to the constraints in C_{T_w}. Then, the attackers represented by $\mathcal{M}_T$ are equivalent to $\Delta(S, R)$.*

Example. Given C_{T_w} for the weakest OAuth attacker above, a minimal unsatisfiable core to specification $\mathcal{S} \equiv C_S \wedge C_{T_w} \wedge C_R$ consists of the following constraints:

$$\begin{aligned} MUC_T \equiv\ & \mathsf{AuthServer} \notin types(\mathsf{Eve}) \wedge \mathsf{Client} \notin types(\mathsf{Eve}) \wedge \\ & \mathsf{alicePwd} \notin init(\mathsf{Eve}) \end{aligned}$$

[2]The details of modeling in Alloy can be found in [27], Chapter 5.

The set of constraints in MUC_T represents an attacker with the following capabilities:

$$types(\mathsf{Eve}) = \{\mathsf{UserAgent}\}$$
$$init(\mathsf{Eve}) = \{\mathsf{eveUID}, \mathsf{evePwd}, \mathsf{aliceUID}, \mathsf{eveCode}, \mathsf{aliceCode}\}$$

This result states that the OAuth 2.0 protocol is robust against any attacker that is at most capable of impersonating a user agent and may have access to the authorization codes. However, against a more powerful attacker, the session integrity of the protocol may no longer be guaranteed. For example, an attacker who is capable of impersonating a client or an authorization server may be able to trick the victim client into associating Alice's session with Eve's account, thus giving the latter access to Alice's account (some of these attacks are described in [37]).

Implementation. As a proof-of-concept, we implemented our approach to computing robustness using the Alloy Analyzer [26], an analysis engine associated with the Alloy language. The analyzer works by translating an input specification of a system to a Boolean satisfiability problem (SAT), which is then solved by an off-the-shelf SAT solver to find a satisfying instance (if it exists).

If the underlying solver is capable of generating an (minimal) unsatisfiable core, the Alloy Analyzer takes advantage of this information by translating the core (which consists of a set of Boolean clauses) back to higher-level constraints in the input Alloy specification. Our approach to computing robustness takes advantage of this feature. Given specification $\mathcal{S}(S, T_w, R)$, a MUC produced by Alloy highlights a minimal subset of the constraints in C_{T_w}; this subset describes the capabilities of one of the attackers in $\Delta(S, R)$.

The Alloy Analyzer, however, currently does not provide a way to enumerate all of MUCs in a specification, which is required to compute a complete representation of $\Delta(S, R)$. To work around this limitation, we employ a naive approach for MUC enumeration by repeatedly invoking the Alloy Analyzer in the following manner: Given the previously generated $MUC_i \subseteq C_{T_w}$, we remove a single *types* or *init* constraint $c \in MUC_i$ from C_{T_w} and attempt to generate another core on the modified specification $\mathcal{S}(S, T'_w, R)$, where $C_{T'_w} = C_{T_w} - \{c\}$. A MUC to this specification, if it exists, would highlight a subset (denoted MUC_j) of constraints in C_{T_w} that may overlap with but differs from MUC_i; these two sets of constraints, MUC_i and MUC_k, represent a pair of incomparable attackers in $\Delta(S, R)$. This process is then repeated until no more MUC is found by the analyzer.

Our model of the OAuth 2.0 protocol was roughly 350 lines of Alloy[3]. On a macOS machine with 2.7 GHz Intel Core i7 and 16 GB of RAM, it took approximately 27 seconds to compute the robustness for the OAuth protocol and the session integrity requirement. Given that OAuth 2.0 is a non-trivial protocol used widely on the web, we believe that the analysis time is reasonable. Furthermore, the performance of this procedure can potentially be improved by leveraging more sophisticated ways of enumerating MUCs [28, 29]; we plan to investigate such techniques as part of future work.

[3]All of the Alloy models for the OAuth example can be accessed on https://github.com/eskang/robustness-for-security.

4.2 Comparing Design Alternatives

As another application of robustness, we show how our approach can be used to compare a pair of alternative system designs in terms of their robustness.

In particular, we compare two different versions of OAuth—OAuth 2.0 and its predecessor, OAuth 1.0. Although 2.0 is intended to be a simpler, more reusable replacement, there has been some debate on whether 2.0 actually improves over 1.0 in terms of security [21]. Since both are designed to provide the same security guarantees, these protocols may appear equally secure if compared with respect to their security requirements (e.g., they share session integrity as a common requirement). Instead, we show that robustness can be used to distinguish these two protocols and argue that one may be more desirable than the other.

OAuth 1.0. Like in 2.0, a typical process in OAuth 1.0 begins with a user initiating a new session with Client (as in Step 1 of Figure 1). Instead of directing the user to AuthServer, however, Client first obtains a *request token* from AuthServer and associates it with the current session. The user is then asked to present the same request token to AuthServer and authorize Client to access their resources. Once notified by the user that the authorization step has taken place, Client exchanges the request token for an access token that can be used subsequently to access their resources.

Robustness for OAuth 1.0. We built an Alloy model of OAuth 1.0 in the same style as for we did OAuth 2.0, and used the approach based on MUCs to compute the robustness of 1.0 for the session integrity requirement. The resulting attacker, again named Eve, has the following capabilities:

$$types(\mathsf{Eve}) = \{\mathsf{UserAgent}, \mathsf{Client}\}$$
$$init(\mathsf{Eve}) = \{\mathsf{eveUID}, \mathsf{evePwd}, \mathsf{aliceUID}, \ldots\}$$

According to our analysis, OAuth 1.0 is robust against an attacker who is capable of impersonating a client *in addition to* a user agent. This suggests that OAuth 1.0 is designed to be more robust than 2.0 against attacks that attempt to deceive a victim user into believing that it is interacting with a trusted client (while, in fact, the client is being impersonated by the attacker).

We provide an informal explanation behind this difference between the two protocols. One of the weaknesses of OAuth 2.0 is that it relies on the user to deliver a correct authorization code (Step 3 in Figure 1) from the authorization server back to the client. This assumption, however, may not necessarily hold in practice; for example, in many browser-based implementations of the client, it has been shown that the user may be deceived into forwarding a wrong authorization code [24], resulting in the user's session being associated with the attacker's authorization code. On the other hand, in OAuth 1.0, a client knows exactly the request token that it expects to receive from the client (i.e., the same one that it obtained from the authorization server at the beginning of the protocol session). Thus, it needs not trust that the user will always deliver a correct request token.

5 RELATED WORK

We are not the first to explore the notion of robustness in the context of secure system design. For example, several researchers have proposed a set of informal guidelines for designing security protocols that are robust against potential attacks [2, 3, 31]. As far as we are aware, however, there has been relatively little work on providing tool support for analysis of system designs for robustness.

Formal modeling and analysis for security (especially protocols) is a well-establish research area, with a number of specification languages (e.g., [1, 7, 15]) and tools (e.g., [6, 12, 30, 34]). Most of these approaches assume some *fixed* model of the attacker—the most common one being the Dolev-Yao model [14], which describes an attacker that can intercept any messages between processes and synthesize new data (except those that are bound by the constraints of cryptography, such as complex passwords or keys). Instead of assuming a fixed attacker model, our work provides a complementary analysis by providing information about the range of attacker capabilities under which the system is capable (or not) of establishing a security requirement.

As mentioned earlier, robustness captures a set of assumptions that the system makes about its environment in order to establish its security requirements. The role of assumptions in security, and vulnerabilities that arise due to a lack of explicit consideration of them, have been studied in the context of system requirements [13], API design [38], and computer security in general [4, 22]. Our work was also inspired by the work of Haley et al. [20], which propose the idea of *trust assumptions* as the set of assumptions about various system components that are needed to establish a requirement, and which advocate that these assumptions be explicitly articulated as early as the requirements stage.

Our approach to computing robustness is similar to the idea of weakest assumption generation in the context of assume-guarantee reasoning [10, 17]. Given system model M and property P, the goal is to generate the *weakest* assumption A for the system environment such that $M \parallel A \models P$. One major difference is that in their approach, behaviors of the system and the environment are specified using labelled-transition systems (LTSs), whereas we explicitly model the malicious environment as the process types and initial knowledge of an attacker. Similarly, Zhang et al. propose a notion of robustness as the set of possible environmental deviations against which the system is capable of establishing a property [40], but again, their notion is defined over LTSs and not designed to model the capabilities of an attacker.

Adversarial robustness is an actively studied topic in machine learning (ML) [5, 8], but is also a very different type of concept than our notion of robustness: It mainly concerns the ability of a ML-based classifier to make correct predictions under adversarial samples.

6 DISCUSSIONS AND FUTURE WORK

Our approach to modeling attackers, based on the process types and knowledge, has a number of limitations that merit further investigation. First, a multi-faceted model that considers not only the actions of the attacker but also costs associated with those actions may be needed to capture more realistic attacker profiles. Second, security vulnerabilities exist across multiple layers of system abstraction (e.g., architecture, code, hardware), and our focus here is mainly on the high-level design and architecture. Finally, the result of our robustness analysis depends on the fidelity of the system model and requirement specification. In practice, building a model that captures all of the system details that are relevant to security is likely to be infeasible and thus, our approach is best suited as a complementary analysis to other secure development activities (such as requirements elicitation, testing, and static analysis).

Our next step is to build on our notion of robustness to develop techniques for *systematically designing robust systems*. Recall that robustness embodies various assumptions that the system makes about its (potentially malicious) environment. Some of these assumptions may be more difficult to establish than others, depending on the application domain. For example, on the Internet, it may not be prudent to assume that the user will never visit a malicious website and get tricked into exposing potentially sensitive data. If an existing system relies on such an assumption, the designer may wish to re-design the system by eliminating or *weakening* the assumption, thus improving its robustness. We plan to explore (semi-)automated approaches to support this redesign process, for example, by leveraging techniques from model transformation and repair [11, 19, 35, 39].

ACKNOWLEDGMENTS

We'd like to thank our anonymous reviewers for their suggestions that helped greatly improve this paper. This work was supported in part by the National Science Foundation awards CCF-1918140 and CNS-1801546.

REFERENCES

[1] Martín Abadi and Andrew D. Gordon. 1997. A Calculus for Cryptographic Protocols: The Spi Calculus. In *CCS '97, Proceedings of the 4th ACM Conference on Computer and Communications Security, Zurich, Switzerland, April 1-4, 1997.* 36–47.
[2] Martín Abadi and Roger M. Needham. 1994. Prudent engineering practice for cryptographic protocols. In *1994 IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, USA, May 16-18, 1994.* 122–136.
[3] Ross J. Anderson and Roger M. Needham. 1995. Robustness Principles for Public Key Protocols. In *Advances in Cryptology - CRYPTO '95, 15th Annual International Cryptology Conference, Santa Barbara, California, USA, August 27-31, 1995, Proceedings.* 236–247.
[4] Helen Armstrong and Matt Bishop. 2005. Uncovering Assumptions in Information Security. In *IFIP TC11 WG 11.8 Forth World Conference "Information Security Education" (WISE4).* Curtin Research Publications, 223–231.
[5] Battista Biggio, Giorgio Fumera, and Fabio Roli. 2014. Security Evaluation of PatternClassifiers under Attack. *IEEE Trans. Knowl. Data Eng.* 26, 4 (2014), 984–996.
[6] Bruno Blanchet. 2001. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 11-13 June 2001, Cape Breton, Nova Scotia, Canada.* 82–96.
[7] Michael Burrows, Martín Abadi, and Roger M. Needham. 1990. A Logic of Authentication. *ACM Trans. Comput. Syst.* 8, 1 (1990), 18–36.
[8] Nicholas Carlini, Anish Athalye, Nicolas Papernot, Wieland Brendel, Jonas Rauber, Dimitris Tsipras, Ian J. Goodfellow, Aleksander Madry, and Alexey Kurakin. 2019. On Evaluating Adversarial Robustness. *CoRR* abs/1902.06705 (2019).
[9] Eric Y. Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. 2014. OAuth Demystified for Mobile Application Developers. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014.* 892–903.
[10] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. 2003. Learning Assumptions for Compositional Verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 331–346. https://doi.org/10.1007/3-540-36577-x_24
[11] Thomas Colcombet and Pascal Fradet. 2000. Enforcing Trace Properties by Program Transformation. In *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000.* 54–66.
[12] Cas J. F. Cremers. 2008. The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings.* 414–418.
[13] Mohammad Torabi Dashti and David A. Basin. 2016. Security Testing Beyond Functional Tests. In *Engineering Secure Software and Systems - 8th International*

Symposium, ESSoS 2016, London, UK, April 6-8, 2016. Proceedings (Lecture Notes in Computer Science), Juan Caballero, Eric Bodden, and Elias Athanasopoulos (Eds.), Vol. 9639. Springer, 1–19.

[14] Danny Dolev and Andrew Chi-Chih Yao. 1983. On the security of public key protocols. *IEEE Trans. Inf. Theory* 29, 2 (1983), 198–207.

[15] F. Javier Thayer Fabrega, Jonathan C. Herzog, and Joshua D. Guttman. 1998. Strand Spaces: Why is a Security Protocol Correct?. In *1998 IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 3-6, 1998, Proceedings*. 160–171.

[16] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2016. A Comprehensive Formal Security Analysis of OAuth 2.0. In *ACM SIGSAC Conference on Computer and Communications Security*. 1204–1215.

[17] D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer. 2002. Assumption generation for software component verification. In *Proceedings - ASE 2002: 17th IEEE International Conference on Automated Software Engineering*. IEEE, 3–12. https://doi.org/10.1109/ASE.2002.1114984

[18] OAuth Working Group. 2013. OAuth 2.0 Threat Model and Security Considerations. https://tools.ietf.org/html/rfc681.

[19] Joshua D. Guttman, Moses D. Liskov, and Paul D. Rowe. 2014. Security Goals and Evolving Standards. In *Security Standardisation Research - First International Conference, SSR 2014, London, UK, December 16-17, 2014. Proceedings*. 93–110.

[20] Charles B. Haley, Robin C. Laney, Jonathan D. Moffett, and Bashar Nuseibeh. 2006. Using trust assumptions with security requirements. *Requir. Eng.* 11, 2 (2006), 138–151.

[21] Eran Hanmer. 2012. OAuth 2.0 and the Road to Hell. https://hueniverse.com/2012/07/26/oauth-2-0-and-the-road-to-hell.

[22] Cormac Herley and Paul C. van Oorschot. 2017. SoK: Science, Security and the Elusive Goal of Security as a Scientific Pursuit. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 99–120.

[23] C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (1978), 666–677.

[24] Egor Homakov. 2012. Most Common OAuth2 Vulnerability. http://homakov.blogspot.ch/2012/07/saferweb-most-common-oauth2.html.

[25] Internet Engineering Task Force. 2014. OAuth Authorization Framework. http://tools.ietf.org/html/rfc6749.

[26] Daniel Jackson. 2006. *Software Abstractions: Logic, language, and analysis*. MIT Press.

[27] Eunsuk Kang. 2016. *Multi-Representational Security Modeling and Analysis*. Ph.D. Dissertation. MIT.

[28] Mark H. Liffiton, Alessandro Previti, Ammar Malik, and João Marques-Silva. 2016. Fast, flexible MUS enumeration. *Constraints An Int. J.* 21, 2 (2016), 223–250.

[29] Mark H. Liffiton and Karem A. Sakallah. 2005. On Finding All Minimally Unsatisfiable Subformulas. In *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings (Lecture Notes in Computer Science)*, Fahiem Bacchus and Toby Walsh (Eds.), Vol. 3569. Springer, 173–186.

[30] Catherine A. Meadows. 1996. The NRL Protocol Analyzer: An Overview. *J. Log. Program.* 26, 2 (1996), 113–131.

[31] Michael K. Reiter and Stuart G. Stubblebine. 1999. Authentication Metric Analysis and Design. *ACM Trans. Inf. Syst. Secur.* 2, 2 (1999), 138–158.

[32] San-Tsai Sun and Konstantin Beznosov. 2012. The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. 378–390.

[33] Emina Torlak, Felix Sheng-Ho Chang, and Daniel Jackson. 2008. Finding Minimal Unsatisfiable Cores of Declarative Specifications. In *FM*. 326–341.

[34] Luca Viganò. 2006. Automated Security Protocol Analysis With the AVISPA Tool. *Electr. Notes Theor. Comput. Sci.* 155 (2006), 61–86.

[35] Eelco Visser. 2001. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In *Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings*. 357–362.

[36] Rui Wang, Shuo Chen, and XiaoFeng Wang. 2012. Signing Me onto Your Accounts through Facebook and Google: A Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*. 365–379.

[37] Rui Wang, Shuo Chen, and XiaoFeng Wang. 2012. Signing Me onto Your Accounts through Facebook and Google: A Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. In *IEEE Symposium on Security and Privacy*. 365–379.

[38] Rui Wang, Yuchen Zhou, Shuo Chen, Shaz Qadeer, David Evans, and Yuri Gurevich. 2013. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*. 399–314.

[39] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*. 356–366.

[40] Changjian Zhang, David Garlan, and Eunsuk Kang. 2020. A Behavioral Notion of Robustness for Software Systems. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.