Finding Software Supply Chain Attack Paths with Logical Attack Graphs

Luís Soeiro[1][0009-0003-8609-1352], Thomas Robert[1][0000-0002-4423-5720], and Stefano Zacchiroli[1][0000-0002-4576-136X]

LTCI, Télécom Paris, Institut Polytechnique de Paris, France
https://www.ip-paris.fr
{luis.soeiro,thomas.robert,stefano.zacchiroli}@telecom-paris.fr

Abstract. Cyberattacks are becoming increasingly frequent and sophisticated, often exploiting the software supply chain (SSC) as an attack vector. Attack graphs provide a detailed representation of the sequence of events and vulnerabilities that could lead to a successful security breach in a system. MulVal is a widely used open-source tool for logical attack graph generation in networked systems. However, its current lack of support for capturing and reasoning about SSC threat propagation makes it unsuitable for addressing modern SSC attacks, such as the XZ compromise or the 3CX double SSC attack. To address this limitation, we propose an extension to MulVal that integrates SSC threat propagation analysis with existing network-based threat analysis. This extension introduces a new set of predicates within the familiar MulVal syntax, enabling seamless integration. The new facts and interaction rules model SSC assets, their dependencies, interactions, compromises, additional security mechanisms, initial system states, and known threats. We explain how this integration operates in both directions and demonstrate the practical application of the extension.

Keywords: software supply chain · logical attack graph · threat propagation · security mechanisms

1 Introduction

Advances in information technology have consistently been shadowed by the proliferation and rising complexity of cyberattacks. The widespread adoption of Free and Open Source Software (FOSS), driven by scientific, industrial, and economic motivations [19], has further expanded the attack surface due to its distributed and resource-constrained development model [12]. Within this context, the Software Supply Chain (SSC) has emerged as a critical target. Its global interconnectedness and limited transparency enable threat actors to exploit vulnerabilities (e.g., Log4Shell) or introduce malicious code, bypassing traditional defenses and propagating attacks across dependent systems [31].

SSC attacks can also be combined. In the 2023 case of the 3CX attack [1], two SSC attacks had to be carried out in a sequence of events. First, the server

that distributed Trading Technologies' software was compromised, leading to the injection of a backdoor in the X_Trader software, which was then available for download. Then, an employee of the 3CX company downloaded the X_Trader software and executed it on his personal computer. The malicious software then helped threat actors connect to the 3CX systems using the employee's authenticated VPN connection. The attackers ultimately compromised the 3CX build environment, injecting malicious code into the signed Windows and macOS versions of the 3CXDesktopApp, which affected the company's customers.

Many models exist to capture threat knowledge and the progression of attacks on a network. Attack trees and attack graphs are used to decompose and understand the steps involved in complex attack scenarios [13]. Moreover, such models can be automatically generated from system introspection or from Cyber Threat Intelligence streams of data [10]. While attack trees capture a single attack goal, attack graphs can capture multiple attack goals [24] and multiple attack paths [10]. The Logical Attack Graph (LAG) formalism introduced in the seminal work of MulVal [18] is widely used and has been regularly extended over the past two decades [29]. However, neither MulVal nor its extensions are prepared to reason about SSC threats [4]. This paper introduces a new MulVal extension that integrates SSC threat propagation reasoning. A full replication package containing all the code presented in this work is available from Zenodo [27].

The following research questions will be answered in this work:

RQ1: *To what extent is it possible to formalize knowledge of SSC attacks into LAG?*

RQ2: *To what extent does such a formalism uncover non-trivial attack scenarios?*

This paper is organized as follows: Section 2 presents related work; Section 3 provides background on MulVal; Section 4 introduces our contribution and approach; Section 5 details how the extension rules integrate with MulVal; Section 6 presents scenarios demonstrating real-world use; Section 7 revisits the research questions; Section 8 discusses limitations and possible mitigations; and Section 9 concludes with closing remarks and future work.

2 Related work

There is substantial research on SSC attack and countermeasure elicitation [12], including work on malware enabling such attacks [17,16] and SSC technical processes [7]. However, these analyses cover only the SSC assets without clear links to the systems that depend on them. The log model [28] proposes threat propagation reasoning for the SSC, but it lacks modeling of available security mechanisms, making its analysis pessimistic, and it does not address the extra complexity of modeling cyberattacks against networked systems. Attack trees and attack graphs generalize complex scenarios [13] and have been applied to SSCs [11]; attack graphs (LAGs) better support multiple goals and paths [10]. To our knowledge only two works attempt to bridge SSC and networked-system

scopes: the Hardening Framework for Substations offers interactive countermeasures but models Software Supply Chain Attacks (SSCA) as a simple Boolean state [4], and CORAL extends MulVal LAGs for container risks [30] yet does not capture the full range of SSCA tampering scenarios.

MulVal cannot compute SSC threat propagation because it lacks predicates for SSC graphs and propagation rules, requires a priori vulnerability declarations (so emergent paths are missed), cannot model vulnerabilities that enable unintended network connections, and has no malicious-software constructs.

3 Background

MulVal is a widely used open-source tool for generating logical attack graphs for networked systems. In these graphs, nodes represent logical statements about system state or attacker capabilities. MulVal models and reasons about those statements using Datalog, a declarative logic language (a safe subset of Prolog) for defining and querying deductive databases. It relies on five concepts: variables, constants, predicates, formulae, facts and inference rules. A predicate formula is an expression $pred_1(t_1, \ldots, t_n)$ where $pred_1$ is the predicate name and t_i are the terms it applies to. A formula declared true is a *fact*; otherwise it is used to define *inference rules*. An inference rule is a Horn clause, $P_0 : -P_1, \ldots, P_n$, such that the predicate formula P_0 is true when the conjunction of $P_1 \wedge \cdots \wedge P_n$ is true. The terms used in the predicate formula can be constants (strings used as identifiers of the problem modeled) or variables. Variables are used to describe the constraints binding the parameters among $P_0, \ldots, P_n$. The deductive database is the combination of the facts and all the inference rules. It can be queried to determine if some predicate formulae are true. The interpreter of these queries can provide the trace of all the rules used. MulVal encodes system state as facts and attacker behaviors as inference rules, so analyses produce attack-graph derivations that show exactly which facts and rules lead to a compromise.

4 A MulVal extension for SSC

We extend MulVal to capture and reason about the core elements of the SSC graph: *host (H)*, *build environment (BE)*, *transformer (T)*, and *software artifact (SA)* [28]. These assets of the SSC depend on each other. The dependencies define an SSC graph, in which the assets are the vertices.

4.1 Approach

In this section, we introduce new predicates to capture SSC assets and their dependencies. Then, we introduce new predicates to capture some previously uncovered aspects of attack behavior. Finally, we introduce predicates and inference rules to capture security mechanisms in the SSC. This work paves the way for Section 5, which presents their integration into MulVal and the resulting threat propagation analysis framework.

To compute the SSC contributions to the threat level on a given vertex e of the SSC graph we need to trace the contributions of all other vertices on paths that reach e. For instance, the threat level of a *software artifact sa* depends on the threat levels of all the SSC vertices that are on paths leading to *sa*, e.g., hosts, build environments, and other software artifacts used as input. Conversely, taking advantage of tools like MulVal to capture usual attacks (e.g., principal compromise, vulnerability exploit) on hosts (either virtual or physical) contributes to a better assessment of threats for those vertices in the SSC.

4.2 Modeling SSC assets and their interactions

The SSC assets (i.e., the SSC-graph core elements), their dependency organization, and their initial known unsafe state (i.e., vulnerable or compromised) are modeled by the newly introduced predicates:

- `vulNetworkProperty(vulID, protocol, port, user)` – binds a vulnerability identifier *vulID* to a network protocol, port, and user. This enables the system to model the case where a vulnerability transforms a piece of software not intended to provide a network service into an access point for a remote attacker (e.g., exposure of the *RMI* protocol on port 1099 in the Log4Shell vulnerability [6]);
- `signed<X>(`key`, `e`)`, $X \in \{C, SA\}$ – declares that key was used to sign certificate or software artifact e.
- `issued(`$Cert_1$`, `$Cert_2$`)` – declares that $Cert_1$ has been used to issue $Cert_2$;
- `compromised<X>(`e`)`, $X \in \{H, BE, T, K, C\}$ – declares that a given element of type X is compromised. H, BE, T, K, and C denote, respectively, *host*, *build environment*, *transformer*, *signing key*, and *certificate*;
- `maliciousSA(`sa`)` – declares that *software artifact sa* is known to be malicious;
- `isolationEscapeBE(`BE`)` – used to infer situations where there is an escape from an isolation mechanism;
- `hosted(`h`,`be`)`, `executed(`be`,`t`)`, `wasInputTo(`sa`,`t`)`, `wasBuildToolTo(`sa`,`t`)`, `wasPresent(`sa`,`h`)`, `generated(`t`,`sa`)`, `wasPublishedTo(`sa`,`h`)`, and `transferred(`sa`, `h`)`, with sa, h, be, and t denoting software artifact, host, build environment (where software builds occur), and transformer (the set of operations that take software artifacts and build tools as input and generate new software artifacts), respectively – derived from the Log Model edges [28]. Figure 1 shows an example of an SSC graph that contains all edge types and all elements.

4.3 Malicious software artifacts

It has been observed that the complexity (e.g., lines of code, number of source files) of malicious software increases roughly at one order of magnitude per decade [2]. The number of malicious software artifacts being uploaded to popular

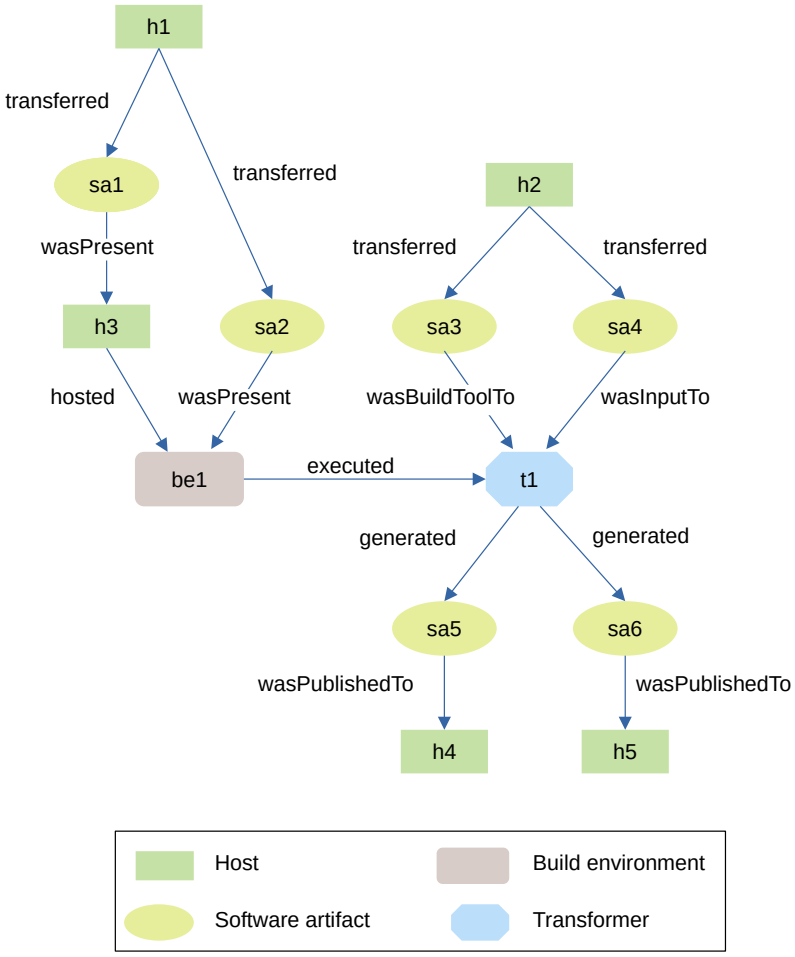

Fig. 1. The software supply chain for software artifacts sa5 e sa6

programming-language repositories (e.g., PyPI, CRAN, npm) has surpassed the number of vulnerable software [23]. While malicious software can have many different behaviors [15], they share one common trait: the ability to autonomously perform actions. That changes the way attack progress is usually modeled in MulVal. Thus, we introduce the `execBatchCode` predicate that models this attack behavior to MulVal's reasoning system. Similarly to `execCode`, we define interaction rules that state the conditions under which the system will autonomously execute malicious code. Listing 1.1 shows one `execBatchCode` interaction rule. If a software artifact *SA* was observed executing on host *Host*, was classified as malicious (determined via `maliciousSA(SA)`), and executed under principal *User* (with `canAccessFile(...)` used to determine the principal under which

Listing 1.1. Interaction rule that shows one effect of a malicious software artifact

```
execBatchCode(Host, SA, User) :-
  wasPresent(SA, Host),
  maliciousSA(SA),
  canAccessFile(Host, User, Access, SA)
```

SA ran), then *SA* can autonomously execute code as principal *User* on host *Host* without human interaction.

4.4 Modeling security mechanisms in SSC

Security mechanisms act as countermeasures for attacks (e.g., a working firewall prevents an outside connection to a vulnerable internal service). In the absence of a reachable privilege-escalation vulnerability or credential theft, the operating-system access-control mechanism prevents a malicious software artifact running as one principal from injecting code into another artifact running as a different principal. When computing possible attack paths for a scenario we consider the preventive nature of the existing security mechanisms that are deployed. This consideration makes the threat-propagation analysis more accurate by removing unreachable attack paths.

MulVal already models many security mechanisms, and our extension takes advantage of them. However, two mechanisms broadly used in SSC are missing in MulVal: build-environment isolation and authenticity verification of software artifacts.

Isolation of build environments We consider existing security mechanisms that can prevent the propagation of threats. Isolation of a build environment prevents the flow of threats from it or to it. For instance, let a *host* H_1 provide isolation for its *build environments* BE_1 and BE_2. Then a malicious *software artifact* running in BE_1 cannot propagate malicious code to the BE_2 asset or to assets that depend on it. Yet, if the isolation mechanism is compromised or if dependencies exist between the assets in BE_2 and those produced in BE_1, propagation can still occur. We define predicates to cover a reasonable set of cases.

Different isolation mechanisms for computer systems are available (e.g., processes, containers, virtualization), each with trade-offs between security and performance overheads [26]. Independently of the underling isolation mechanism, we model the possible isolation outcomes using the concepts of isolated build environment and MulVal access control. We show two scenarios. In the first, there is no isolation of the *build environment* be_1; only access control is used to prevent one build run from interfering with other build runs on the same *host* h_1. In this case we model it by declaring a single be_1 in h_1, `hosted(`h_1`,` be_1`)`, and one *transformer* t_i for each build run that is executed. Let N be the number of independent builds. We declare the predicates `executed(`be_1`,` t_i`)` and `localFileProtection(`be_1`,` $user_i$`,` $access_i$`,` pSA_i`)` for $i \in \{1 \dots N\}$,

Listing 1.2. Interaction rule that shows the conditions for an escape of the build-environment isolation

```
isolationEscapeBE(BE) :-
  execBatchCode(BE, SA, User),
  wasPresent(VulnSA, BE),
  vulExists(BE, _, VulnSA, localExploit, isolationEscape)

isolationEscapeBE(BE) :-
  execBatchCode(BE, SA, User),
  hosted(H, BE),
  wasPresent(VulnSA, H),
  vulExists(H, _, VulnSA, localExploit, isolationEscape)
```

where $user_i$, $access_i$, and pSA_i are, respectively, the principal, the access type (e.g., read, write), and the logical path of the *software artifacts* used by each build run.

The second scenario aligns with current expectations of isolation that come from using build platforms. Users are relying more on services that offer Continuous Integration/Continuous Deployment (CI/CD) workflows for building software [22]. In this scenario, each build environment is isolated from the others. We model it by declaring multiple *build environments*, each with only one *transformer*. Let h_1 be the host and N be the number of independent builds. We declare the predicates `hosted(`h_1`,` be_i`)` and `executed(`be_i`,` t_i`)` for $i \in \{1 \dots N\}$. We assume that each be_i is isolated.

Since vulnerabilities may allow for process escape (i.e., privilege escalation), container escape [14] or virtualization escape [20], we add new rules to MulVal to capture those interactions in the context of the SSC. We define the predicate `isolationEscapeBE(`BE`)` to cover both virtualization and container escape. Listing 1.2 shows two interaction rules that allow modeling the situation where a malicious software artifact escapes from the isolating container or virtual machine. The first rule is triggered by a vulnerable software artifact located inside the *build environment* (container or virtual machine) and the second rule is triggered by an artifact located on the *host* that hosted the *build environment*. For both escapes to succeed, there must be a software artifact that has received the propagation of a vulnerability with the property `vulProperty(vulID, localExploit, isolationEscape)`.

Authenticity of software artifacts Several SSC attack paths rely on hijacking the secure dissemination of software [11]. To improve software dissemination, distribution systems either started to rely on digital signatures [3] or proposed them [9,4]. Modeling authentication mechanisms within the SSC is complex; to keep the analysis tractable, we model attacks limited to software-artifact tampering and trust-chain compromises. Because signing adoption varies widely [25], we account for cases where data authenticity is enforced or absent for different objects. For example, a system may obtain software from official repositories,

Listing 1.3. Interaction rule that shows SA vulnerability propagation

```
compromisedK(PrivateKey) :-
  compromisedH(H),
  wasPresent(PrivateKey, H)

compromisedC(Certificate) :-
  compromisedK(PrivateKey),
  signedC(PrivateKey, Certificate)

maliciousSA(SA) :-
  compromisedC(Certificate),
  validateSA(Certificate, SA)
```

verified by the operating-system package manager, or from unverified sources (e.g., PyPI).

Data authentication relies heavily on certificates and signing keys. They build a trust chain from a root of trust through root anchors up to the certificate used to validate a software artifact. Yet, threat actors are compromising code-signing mechanisms to distribute malicious software as legitimate (e.g., XZ, SolarWinds, 3CX) [8]. We introduce rules to identify the effects of key compromises at any stage of trust chains. Modeling certificate chains is done through the predicate `issued(`$Cert_1$`,` $Cert_2$`)`. It captures trust dependencies along the chain. We identify keys and signed objects with the predicates `signedSA(`Key`,` Sa`)` (signing *software artifacts*) and `signedC(`Key`,` $Cert$`)` (signing *certificates*). This allows the rules to identify the effects of key compromises at any stage of the trust chain. We introduce the predicate `validateSA(`$Cert$`,` Sa`)` to declare that authenticity is checked for Sa using the public key bound to $Cert$ along the SSC (e.g., for all packages from a Debian GNU/Linux distribution). These predicates are sufficient to cover the basics of SSC data-authenticity mechanisms.

Compromises of private keys or corresponding certificates are defined using the predicates `compromisedK(`key`)` and `compromisedC(`$Cert$`)`. Our extension then considers all *software artifacts* that were signed by a compromised private key as malicious and propagates the consequences. Listing 1.3 shows some of the rules that account for violations of privacy or integrity of signing keys. The first rule states that the private key $PrivateKey$ is compromised if it was stored on a compromised *host*. The second rule states that a certificate signed by a compromised key is also compromised. The third rule states that a software artifact signed with a compromised key is compromised.

5 Integration of SSC threat propagation with MulVal

We introduced new predicates to capture SSC assets, their dependencies, and their threat states. Yet, we need to introduce new predicates that bridge our new rules and MulVal's existing rules.

Listing 1.4. Interaction rule that shows SA vulnerability propagation

```
vulnerableSA(SA, VulID) :-
  vulnerableSA(SA_input, VulID),
  wasInputTo(SA_input, T),
  generated(T, SA)

vulnerableSA(SA, VulID) :-
 vulExists(Host, VulID, SA)
```

5.1 Vulnerable software propagation

We introduce the predicate `vulnerableSA(...)` to encode vulnerability-inference rules derived from SSC interactions. Consider Figure 1. Let $sa4$ be a vulnerable Java-language software artifact (e.g., the Log4J library version 2.14.1, which contains the Log4Shell vulnerability [6]). It *was input* to the *transformer* $t1$, which generated *software artifacts* $sa5$ and $sa6$. Listing 1.4 shows two rules for vulnerability propagation. The first rule states that a SA is vulnerable if it was generated by the *transformer* T and T used a vulnerable SA as input. The second states that an SA is vulnerable if it was declared vulnerable in the initial state (e.g., `vulExists(h2, vulLog4Shell, sa4).`). The rules allow the system to infer an arbitrarily long chain of SSC vertices that propagate vulnerable *software artifacts* (i.e., `vulnerableSA(...)` appears on both the left and right sides of the first rule).

An automated SCA analysis of $sa5$ and $sa6$ would detect the presence of the library $sa4$ in this case because, in Java, the binary library dependencies are copied to the resulting binary software package. However, this is not always the case. For other scenarios where the dependency is statically linked into the generated binaries, simple scanning for artifacts will not identify the original libraries. Let $sa4$ be a vulnerable C-language software artifact (e.g., the OpenSSL library version 1.0.1, which contains the Heartbleed vulnerability [5]) that is compiled, statically linked, and included by the *transformer* $t1$ in the binary code of $sa5$ and $sa6$. As in the previous case, the extension would also infer that $sa5$ and $sa6$ are vulnerable and use this information to further propagate threats.

The effects of vulnerability propagation in the SSC are perceived when the affected software artifacts are executed. Then, their vulnerabilities are ready to be exploited. Listing 1.5 shows some of the rules that allow MulVal to reason about inferred or declared vulnerabilities of software artifacts that come from the SSC. The first inference rule states that if there was a vulnerable software artifact SA present (i.e., observed to be executing) on host $Host$, then MulVal's `vulExists(...)` is true. This allows MulVal rules to infer its consequences.

In the second inference rule of Listing 1.5, the extension signals to MulVal the outcome of the vulnerable software artifact SA found on host $Host$ by SSC vulnerability propagation. The effect of the vulnerability is the provision of a network service (i.e., it is ready to receive network connections) on port $Port$,

Listing 1.5. Interaction rule that shows SSC vulnerability propagation as input to MulVal reasoning rules

```
1  vulExists(Host, VulID, SA, Range, Consequence) :-
2    vulnerableSA(SA, VulID),
3    wasPresent(SA, Host),
4    vulProperty(VulID, Range, Consequence)
5
6  networkServiceInfo(Host, SA, Protocol, Port, User) :-
7    vulnerableSA(SA, VulID),
8    vulNetworkProperty(VulID, Protocol, Port, User),
9    wasPresent(SA, Host),
10   vulProperty(VulID, remoteExploit, privEscalation)
```

Listing 1.6. Interaction rules for SSC compromise propagation

```
1  compromisedH(H) :- maliciousSA(SA),  wasPresent(SA, H)
2  compromisedBE(BE) :- compromisedH(H), hosted(H, BE)
3  compromisedT(T, BE) :- compromisedBE(BE), executed(BE, T)
4  maliciousSA(SA) :- compromisedT(T, BE),  generated(T, SA)
5
6  compromisedT(T, BE) :-
7    executed(BE, T),
8    execBatchCode(BE, SA, User),
9    canAccessFile(BE, User, write, SA_build),
10   wasBuildToolTo(SA_build, T)
11
12 principalCompromised(Victim) :-
13   hasAccount(Victim, H, User),
14   compromisedH(H)
15
16 compromisedH(H) :-  execCode(H, root)
```

with the access privileges of $User$. This allows MulVal to reason about other conditions (e.g., network access permitted) and generate an attack path that depends on having the network service available.

5.2 Propagation of malicious software and asset compromises

We complement the dynamic and static mechanisms of malicious software detection [21] with inference. Given a set of known compromised elements in the initial state, this extension infers its effects for threat propagation. The resulting attack paths include inferred malicious software artifacts and asset compromises.

Consider Figure 1. Let $sa1$ be a malicious *software artifact*. The extension will infer by propagation that $\{h3, be1, t1\}$ are compromised and that $\{sa5, sa6\}$ are malicious. Listing 1.6 shows some of the inference rules for compromise propagation. On line 1, the rule states that a host H is compromised if there is a

Listing 1.7. MulVal predicates for defining the initial state

```
1 attackerLocated(internet).
2 hacl(internet, h1, tcp, 443).
3 vulExists(h1, 'CVE-2021-41773', httpd).
4 vulProperty('CVE-2021-41773', remoteExploit, privEscalation).
5 networkServiceInfo(h1, httpd, tcp, 443, user_apache).
6 vulExists(h1, 'CVE-2021-3560', polkit).
7 vulProperty('CVE-2021-3560', localExploit, privEscalation).
```

malicious *software artifact* SA executing on it. On line 2, the rule states that a *build environment* BE is compromised if the *host* H that executed it is compromised. On line 3, the rule states that a *transformer* T is compromised if it was executed by the compromised *build environment* BE. On line 4, the rule states that a *software artifact* SA is malicious if it was generated by a compromised *transformer* T. In the example of Figure 1, $\{sa5, sa6\}$ are malicious because of the rules on lines 1-3.

Line 6 of Listing 1.6 shows an attack path where malicious code compromises the build tool of a build step (a *transformer*). The rule states that a *transformer* T is compromised if it was executed by a *build environment* BE, there was a malicious code SA executing on BE, with principal $User$ (see 1.1) and write access to SA_{build}, which was a build tool to T. For an example, consider Figure 1. Let $sa2$ be the only malicious *software artifact* in the initial state. Then the extension will infer that $\{sa5, sa6\}$ (generated by $t1$) are malicious if $sa2$ has *write* access to $sa3$ (the build tool used by the transformer $t1$).

On line 12 of Listing 1.6, we show a rule that connects SSC threat propagation to MulVal rules. The principal $Victim$ is compromised if it has an account on *host* H which is compromised according to SSC compromise-propagation rules.

Finally, on line 16 of Listing 1.6, we show a rule that connects MulVal inference rules to SSC threat-propagation rules. It states that the *host* H is also compromised if there is a successful attack path leading to H, according to MulVal rules (i.e., `execCode(...)`). We can now present usage scenarios in Section 6.

6 Detecting real-world SSC attacks

To use the extension, we encode the SSC graph and the initial state with logic predicates. The MulVal extension then generates the attack graph using both the existing MulVal predicates and the new extension predicates. In the scenarios shown, the attack paths cannot be found with either MulVal or SSC threat-propagation knowledge alone, because each predicate set covers a different subset of attack steps. These differences are illustrated by the color coding in the attack graphs.

In the first scenario, the attack graph is initiated by a cyberattack on host $h1$ and then SSC threat-propagation rules compute the subsequent effects. We define the SSC by using predicates from those introduced in Section 4.2 for

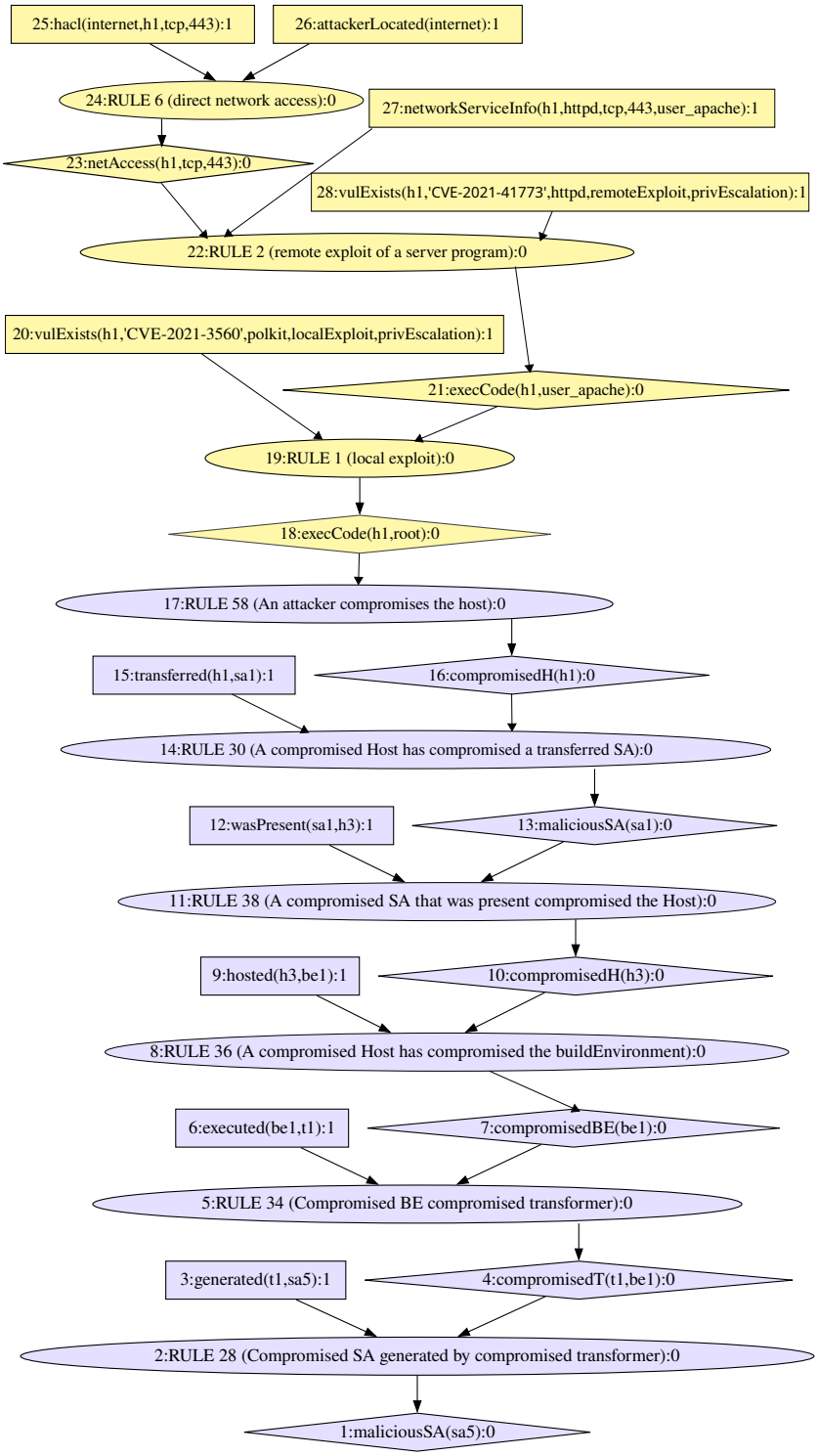

Fig. 2. A pruned attack graph generated for the SSC of $sa5$.

each edge of Figure 1. For example: *transferred(h1, sa1)*, *wasPresent(sa1, h3)*, *hosted(h3, be1)*, *wasPresent(sa2, be1)*, *executed(be1, t1)*, *wasBuildToolTo(sa3, t1)*, *generated(t1, sa5)*. The initial state is shown in Listing 1.7. Apache *httpd* software on host $h1$ is vulnerable to remote access. Additionally, the *polkit* software on $h1$ contains a vulnerability that allows privilege escalation. The resulting attack graph (only shown for $sa5$) appears in Figure 2. Vertices 1-17 (purple) denote new SSC threat-propagation rules and vertices 18-26 (orange) denote existing MulVal rules. There is an attack path that leads, in the first steps, to the root compromise of *host* $h1$. From that point, *software artifact* $sa1$ is inferred to be malicious, which leads to the compromise of *host* $h3$, which compromises both the *build environment* $be1$ and the *software artifact* $sa4$. Finally, both paths lead to the compromised *transformer* $t1$, which leads to the malicious *software artifacts* $sa5$ and $sa6$.

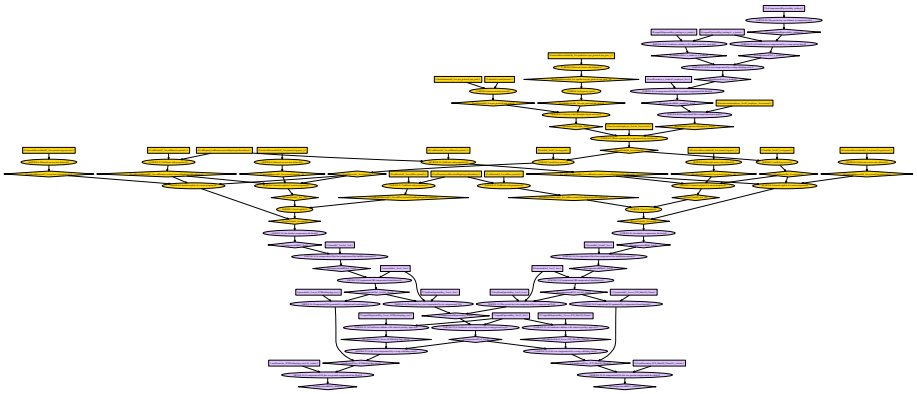

Fig. 3. Intertwined rules (orange and purple) for the 3CX attack graph

In the second scenario, the attack graph for the 3CX double-SSC attack is depicted in Figure 3, color-coded as the previous scenario. Although individual rule text is unreadable at this scale, the dense structure and mixed colors convey both the complexity of the rule set and the tight interdependence between standard attack rules and the proposed SSC propagation rules. The replication package [27] contains 20 additional usage scenarios, including signing-key compromise, build-environment isolation and escape, and combined SSC attacks.

7 Discussion

RQ1: *To what extent is it possible to formalize knowledge of SSC attacks into LAG?* The SSC MulVal extension allows users to account for SSC attacks, even in long chains of interactions, when generating attack paths. It can be used to prioritise the resources that appear on the attack paths for further investigation.

In the example shown in Section 6, the system infers that the software artifacts $\{sa5, sa6\}$ at the end of the SSC are malicious. The attack path begins with an attacker exploiting two vulnerabilities in a remote host. The example shows that the capability of the logical graph generator was correctly expanded to also account for the effects of attacks on the SSC.

RQ2: *To what extent does such a formalism uncover non-trivial attack scenarios?* Because of the inference rules shown, especially in Section 5, threats can be inferred instead of only detected with external tools. An inferred threat makes it possible to reason about its effects on the other elements of the SSC and networked systems, covering complex scenarios. In the best-case scenario (i.e., the inference is effective), the inferred threats in the attack paths can be neutralized (e.g., a new firewall rule that blocks a host from receiving network connections, or a software-artifact version changed). In the worst-case scenario, the resources on the attack paths are all false positives. In this case, the effort of investigating possible compromises is restricted to the resources in the attack paths, a fraction of all the resources available.

8 Limitations

We chose a widely-used FOSS tool for LAG generation. However, there might be other developments that could make the work of integrating SSC knowledge easier. We chose to implement the extension by only adding predicates to the base MulVal rules. In this way, they should be compatible with other extensions. However, if other extensions replace the original MulVal rules (instead of only adding new rules) it might cause integration problems. Despite the absence of facts and inference rules that are specific to FOSS, the need to declare the SSC structure may be an issue for non-free projects. In this case, the original MulVal approach can still be used at the expense of accuracy.

Table 1. Execution time for increasingly larger scenarios.

#Hosts	#SA	#Predicates	Time
3K	39K	4M	53 s
3K	183K	21M	13 min
6K	186K	40M	48 min
15K	195K	100M	4 h 46 min

Dealing with very large graphs can pose scalability problems. MulVal can handle millions of predicates (vertices in the SSC). However, when full SSC graphs are used—because all software artifacts observed on each build environment and host must be represented—the reasoning engine may reach MulVal's limits. We generated scenarios with increasingly larger SSC graphs to gain insight into the possible limits. For the experiment we assumed each host and build environment contains 1,000 to 5,000 unique software packages drawn from a limited

set of operating systems. We executed MulVal with our extension on a computer equipped with an Intel Core i7-12700H CPU and 32 GB of RAM. The results are shown in Table 1. The columns list the number of unique hosts, unique software artifacts, resulting number of predicates, and total running time for logical attack graph generation. We stopped after completing the scenario with 100 million predicates (just under five hours of execution). We observed that the current implementation uses only a single thread for computation. Expanding parallelism is one approach to improve the engine's performance. Another possible solution for supporting even larger SSC graphs is to partition threat-propagation runs around each software artifact and cache results in a network-reachable database.

9 Conclusion

This paper presents an extension to MulVal by introducing new predicates and rules to: (i) model SSC assets and their interactions along which attacks can propagate; (ii) represent assets' security status (e.g., vulnerable, malicious, or compromised); (iii) encode initial knowledge about vulnerable or compromised hosts and software artifacts; and (iv) model security mechanisms and incorporate them into SSC threat propagation. This extension captures complex attack scenarios that combine SSCs and traditional networked-system attacks. Those scenarios display strong interleaving between attack types, indicating that threat identification would not be possible with either reasoning approach alone.

Future work. We plan to develop a mechanism for partitioning, caching, updating, and retrieving partial SSC threat-propagation runs to guarantee scalability for very large graphs. Another area of work is the automatic generation of MulVal input rules. We consider the usage of hardware mechanisms (e.g., Trusted Platform Module) to help instrumentation logic capture running software-artifact information during builds.

Acknowledgments. Supported by the industrial chair Cybersecurity for Critical Networked Infrastructures (cyberCNI.fr) with support of the FEDER development fund of the Brittany region, France.

References

1. Security update thursday 20 april 2023 – initial intrusion vector found. `https://www.3cx.com/blog/news/mandiant-security-update2/`, accessed: 2024-12-26
2. Calleja, A., Tapiador, J., Caballero, J.: A Look into 30 Years of Malware Development from a Software Metrics Perspective, pp. 325–345. Springer International Publishing (2016). `https://doi.org/10.1007/978-3-319-45719-2_15`
3. Catuogno, L., Galdi, C., Persiano, G.: Secure dependency enforcement in package management systems. IEEE Transactions on Dependable and Secure Computing **17**(2), 377–390 (Mar 2020). `https://doi.org/10.1109/tdsc.2017.2777991`
4. Duman, O., Tabiban, A., Wang, L., Debbabi, M.: Measuring and improving the security posture of iec 61850 substations against supply chain attacks. IEEE Transactions on Instrumentation and Measurement **73**, 1–20 (2024). `https://doi.org/10.1109/tim.2024.3400328`

5. Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., Weaver, N., Adrian, D., Paxson, V., Bailey, M., Halderman, J.A.: The matter of heartbleed. In: Proceedings of the 2014 Conference on Internet Measurement Conference. pp. 475–488. IMC '14, ACM (Nov 2014). https://doi.org/10.1145/2663716.2663755
6. Everson, D., Cheng, L., Zhang, Z.: Log4shell: Redefining the web attack surface. In: Proceedings 2022 Workshop on Measurements, Attacks, and Defenses for the Web. MADWeb 2022, Internet Society (2022). https://doi.org/10.14722/madweb.2022.23010
7. Hammi, B., Zeadally, S., Nebhen, J.: Security threats, countermeasures, and challenges of digital supply chains. ACM Computing Surveys **55**(14s), 1–40 (Jul 2023). https://doi.org/10.1145/3588999
8. Ji, T., Fang, B., Cui, X., Wang, T., Zhang, Y., Gu, F., Zheng, C.: Scrutinizing code signing: A study of in-depth threat modeling and defense mechanism. IEEE Internet of Things Journal **11**(24), 40051–40069 (Dec 2024). https://doi.org/10.1109/jiot.2024.3450272
9. Kalu, K.G., Singla, T., Okafor, C., Torres-Arias, S., Davis, J.C.: An industry interview study of software signing for supply chain security. arXiv preprint arXiv:2406.08198 (2024)
10. Konsta, A.M., Lluch Lafuente, A., Spiga, B., Dragoni, N.: Survey: Automatic generation of attack trees and attack graphs. Computers & Security **137**, 103602 (Feb 2024). https://doi.org/10.1016/j.cose.2023.103602
11. Ladisa, P., Plate, H., Martinez, M., Barais, O.: Sok: Taxonomy of attacks on open-source software supply chains. In: 2023 2023 IEEE Symposium on Security and Privacy (SP) (SP). pp. 167–184. IEEE Computer Society, Los Alamitos, CA, USA (may 2023). https://doi.org/10.1109/SP46215.2023.00010, https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00010
12. Ladisa, P., Ponta, S.E., Sabetta, A., Martinez, M., Barais, O.: Journey to the center of software supply chain attacks. IEEE Security & Privacy **21**(6), 34–49 (Nov 2023). https://doi.org/10.1109/msec.2023.3302066
13. Lallie, H.S., Debattista, K., Bal, J.: A review of attack graph and attack tree visual syntax in cyber security. Computer Science Review **35**, 100219 (Feb 2020). https://doi.org/10.1016/j.cosrev.2019.100219
14. Lin, X., Lei, L., Wang, Y., Jing, J., Sun, K., Zhou, Q.: A measurement study on linux container security: Attacks and countermeasures. In: Proceedings of the 34th Annual Computer Security Applications Conference. pp. 418–429. ACSAC '18, ACM (Dec 2018). https://doi.org/10.1145/3274694.3274720
15. Lindorfer, M., Di Federico, A., Maggi, F., Comparetti, P.M., Zanero, S.: Lines of malicious code: insights into the malicious software industry. In: Proceedings of the 28th Annual Computer Security Applications Conference. pp. 349–358. ACSAC '12, ACM (Dec 2012). https://doi.org/10.1145/2420950.2421001
16. Martínez, J., Durán, J.M.: Software supply chain attacks, a threat to global cybersecurity: SolarWinds' case study. International Journal of Safety and Security Engineering **11**(5), 537–545 (oct 2021). https://doi.org/10.18280/ijsse.110505
17. Ohm, M., Plate, H., Sykosch, A., Meier, M.: Backstabber's knife collection: A review of open source software supply chain attacks. In: Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 23–43. Springer International Publishing (2020). https://doi.org/10.1007/978-3-030-52683-2_2
18. Ou, X., Govindavajhala, S., Appel, A.W., et al.: Mulval: A logic-based network security analyzer. In: USENIX security symposium. vol. 8, pp. 113–128. Baltimore, MD (2005)

19. Paschali, M.E., Ampatzoglou, A., Bibi, S., Chatzigeorgiou, A., Stamelos, I.: Reusability of open source software across domains: A case study. Journal of Systems and Software **134**, 211–227 (Dec 2017). https://doi.org/10.1016/j.jss.2017.09.009
20. Pearce, M., Zeadally, S., Hunt, R.: Virtualization: Issues, security threats, and solutions. ACM Computing Surveys **45**(2), 1–39 (Feb 2013). https://doi.org/10.1145/2431211.2431216
21. Polamarasetti, A.: Research developments, trends and challenges on the rise of machine learning for detection and classification of malware. In: 2024 International Conference on Intelligent Computing and Emerging Communication Technologies (ICEC). pp. 1–5. IEEE (Nov 2024). https://doi.org/10.1109/icec59683.2024.10837413
22. Rostami Mazrae, P., Mens, T., Golzadeh, M., Decan, A.: On the usage, co-usage and migration of ci/cd tools: A qualitative analysis. Empirical Software Engineering **28**(2) (Mar 2023). https://doi.org/10.1007/s10664-022-10285-5
23. Ruohonen, J., Saddiqa, M.: A time series analysis of malware uploads to programming language ecosystems (2025). https://doi.org/10.48550/ARXIV.2504.15695
24. Saint-Hilaire, K.A., Neal, C., Cuppens, F., Boulahia-Cuppens, N., Bassi, F.: Attack-defense graph generation: Instantiating incident response actions on attack graphs. In: 2024 IEEE 23rd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). pp. 295–305. IEEE (Dec 2024). https://doi.org/10.1109/trustcom63139.2024.00063
25. Schorlemmer, T.R., Kalu, K.G., Chigges, L., Ko, K.M., Ishgair, E.A., Bagchi, S., Torres-Arias, S., Davis, J.C.: Signing in four public software package registries: Quantity, quality, and influencing factors. In: 2024 IEEE Symposium on Security and Privacy (SP). pp. 1160–1178. IEEE (May 2024). https://doi.org/10.1109/sp54263.2024.00215
26. Shu, R., Wang, P., Gorski III, S.A., Andow, B., Nadkarni, A., Deshotels, L., Gionta, J., Enck, W., Gu, X.: A study of security isolation techniques. ACM Computing Surveys **49**(3), 1–37 (Oct 2016). https://doi.org/10.1145/2988545
27. Soeiro, L., Robert, T., Zacchiroli, S.: Replication package for: Finding software supply chain attack paths with logical attack graphs (2025). https://doi.org/10.5281/zenodo.15924456
28. Soeiro, L., Robert, T., Zacchiroli, S.: Assessing the threat level of software supply chains with the log model. In: 2023 IEEE International Conference on Big Data (BigData). IEEE (Dec 2023). https://doi.org/10.1109/bigdata59044.2023.10386091
29. Tayouri, D., Baum, N., Shabtai, A., Puzis, R.: A survey of mulval extensions and their attack scenarios coverage. IEEE Access **11**, 27974–27991 (2023). https://doi.org/10.1109/access.2023.3257721
30. Tayouri, D., Sgan Cohen, O., Maimon, I., Mimran, D., Elovici, Y., Shabtai, A.: Coral: Container online risk assessment with logical attack graphs. Computers & Security **150**, 104296 (Mar 2025). https://doi.org/10.1016/j.cose.2024.104296
31. Williams, L., Benedetti, G., Hamer, S., Paramitha, R., Rahman, I., Tamanna, M., Tystahl, G., Zahan, N., Morrison, P., Acar, Y., Cukier, M., Kästner, C., Kapravelos, A., Wermke, D., Enck, W.: Research directions in software supply chain security. ACM Transactions on Software Engineering and Methodology **34**(5), 1–38 (May 2025). https://doi.org/10.1145/3714464