# Confusion de Type en C++ :
## État de l'Art et Difficultés de Détection

Florent Saudel
Amossys et IRISA
Email: Florent.Saudel@irisa.fr

Sandrine Blazy
Université Rennes 1, IRISA
Email: Sandrine.Blazy@irisa.fr

Frédéric Besson
Inria
Email: Frederic.Besson@inria.fr

***Abstract*—Le langage C++ s'est imposé comme une référence dans les domaines où la modularité du développement ne doit pas empiéter sur les performances du logiciel final. Les principaux navigateurs, les interpréteurs et même certaines parties du système d'exploitation de Microsoft utilisent le langage C++. L'étude des bases de données de vulnérabilités montre que ces logiciels sont sujets à une catégorie de vulnérabilités particulières, les confusions de type, qui sont tout aussi exploitables que les vulnérabilités plus connues.**

**Cet article présente les mécanismes à l'origine des confusions de type et dresse un état de l'art des méthodes servant à les détecter dans du code source ou du binaire seul. Il met aussi en avant les principales difficultés que rencontrent les analyses de binaire, et propose les grandes lignes d'une nouvelle approche pour détecter des confusions de type dans du binaire.**

## I. Introduction

La modularité apportée par la programmation orientée objet a fait le succès du langage C++. L'héritage permet de mettre en relation plusieurs classes et partager ainsi des données et des comportements. La conversion entre types est le mécanisme donné aux développeurs pour choisir un niveau d'abstraction dans la hiérarchie de classes. Les conversions de types abondent donc naturellement au sein des logiciels utilisant une telle abstraction. Cependant, ce mécanisme est dangereux lorsque la conversion de types ne respecte pas l'ordre établi dans la hiérarchie de classes. Le langage C++ ne vérifie pas dynamiquement les conversions de type de façon automatique. En effet, les types disparaissent après compilation. Aussi, il se peut que la représentation d'une donnée au cours d'une exécution ne corresponde pas à celle attendue par le programme. Lorsqu'elle est provoquée par un attaquant, cette erreur d'interprétation peut compromettre la sécurité d'un programme.

Un exemple marquant est la vulnérabilité référencée CVE-2013-0912 qui a permis de compromettre le navigateur Chrome [1]. Seule cette vulnérabilité a été utilisée pour chaque étape de l'attaque, de la fuite d'information pour outrepasser des contre-mesures, jusqu'au détournement du flot de contrôle vers un *shellcode* malveillant.

Le standard C++ définit une conversion de types comme valide si elle suit un chemin de l'arbre représentant la hiérarchie de classes. La Figure 1 montre un un tel arbre, que nous utilisons pour présenter les trois possibilités de conversions de types. Supposons qu'une conversion de types de F1 (fils de P) vers F2 (autre fils de P) a lieu. Comme il n'existe pas de chemin de F1 à F2, cette conversion est invalide. De plus, toute conversion de types de F1 vers P est valide. En effet, F1 est un fils de P, et donc le programme peut manipuler F1 comme étant du type de P. Une telle conversion d'une classe fille vers une classe parente est appelé *upcast*. Elle est toujours valide et donc sûre, aussi les compilateurs la réalisent de façon implicite.

Une conversion de types dans l'autre sens, de P vers F1 par exemple, est appelée *downcast*. Ici, cette conversion peut poser problème car P possède deux fils. Pour être valide, la conversion doit respecter un invariant: pour toute exécution, le type dynamique à convertir doit être F1. Pour vérifier cette invariant, il est nécessaires de retrouver l'ensemble des types dynamiques vers lequel peut pointer chaque pointeurs, Pande [2] a démontré que ce problème est NP-complet. Par conséquent les compilateurs ne cherchent pas à s'assurer pas de la validité de cet invariant et un *downcast* n'est donc pas sûr.

En C++, la conversion d'un type se fait au moyen de trois opérateurs. Ils permettent au développeur de choisir le compromis qu'il souhaite entre sûreté et performance. Nous présentons ces conversions en partant de la moins sûre.

- `reinterpret_cast`: aucune vérification n'est effectuée sur la compatibilité du type source et du type de destination, les bits de données sont juste réinterprétés selon le nouveau type. Il est donc possible de convertir un objet F1 vers F2 avec cette conversion.
- `static_cast`: le compilateur s'assure seulement qu'il existe une relation de parenté entres les types. Il doit exister un chemin entre le type statique à convertir et le type d'arrivée, dans n'importe quel sens (*i.e. upcast* ou *downcast*). C'est au développeur de s'assurer du respect de l'invariant.
- `dynamic_cast`: le compilateur s'assure de la parenté entre les types et injecte au sein du code une vérification dynamique. Son rôle est de vérifier lors de l'exécution du programme que le type source dynamique est véritablement compatible avec le type de destination (cas du *downcast*). Si la conversion est invalide, le `dynamic_cast` renvoie un pointeur nul.

Dans le cas d'un *downcast* de P vers F1 (*cf.* Figure 1), un `static_cast` ne suffit pas pour s'assurer de la validité de la conversion. Néanmoins à la compilation, le `static_cast` est correct et ne lève aucune alerte car il existe bien un arc entre P et F1. Malheureusement, le développeur s'est trompé

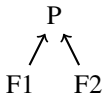


Figure 1. Exemple de hiérarchie de classe

à propos de l'invariant : comme P a deux types fils, il se peut que lors d'une exécution, l'objet dynamique soit de type F2. La conversion devient donc invalide et illustre un exemple de confusion de type.

Le `dynamic_cast` est la conversion la plus sûre mais elle possède un impact sur les performances qui est difficilement prévisible. Il dépend à la fois de l'implémentation fournie par la bibliothèque standard et de la taille de la hiérarchie de classes [3]. Ceci explique en partie son impopularité au sein d'une communauté de développeurs où la performance est primordiale. Par ailleurs, il s'avère que pour certains logiciels utilisant énormément de conversions, comme les navigateurs, l'impact sur les performances est tel que l'utilisation du `dynamic_cast` est interdite.

Dans la suite nous dressons un état de l'art des techniques permettant la détection de cette vulnérabilité. Nous commençons par les méthodes opérant sur le code source, puis nous considérons les méthodes s'appliquant directement au binaire. Des difficultés supplémentaires apparaissent alors pour le binaire, ce qui empêchent l'utilisation de techniques ayant fait leur preuve pour le code source. Enfin, nous mettons en relief l'écart existant entre les deux approches et proposons une nouvelle direction pour l'analyse de confusion de type.

## II. Analyse de code source

Plusieurs outils intégrés au sein des compilateurs GCC et Clang, nommés *sanitizers* visent à détecter les occurrences de certains bugs lors de l'exécution d'un programme. L'intérêt principal d'un *sanitizer* est de lever une alerte juste avant que le bug ne se produise, ce qui permet alors d'identifier son origine exacte. Par ailleurs, ce procédé permet de détecter des bugs même s'ils ne sont pas fatals au programme, permettant donc potentiellement de trouver un plus grand nombre de bugs qu'avec une exécution normale du programme. Dans la suite, nous considérons uniquement les *sanitizers* visant la détection de confusions de type. Les trois principaux sont Ubsan [4], CaVeR [5], et TypeSan [6].

Tous les *sanitizers* suivent une même approche combinant une analyse statique de code source et une vérification dynamique. Un premier composant, prenant la forme d'une passe de compilation, localise les conversions potentiellement dangereuses. Ensuite une vérification est injectée juste avant le code réalisant la conversion; elle s'effectuera à chaque exécution de cette partie du code. Comme elle est dynamique, cette vérification opère sur le code assembleur qui n'est pas typé. Par conséquent des informations supplémentaires sont nécessaires; le *sanitizer* rajoute donc des métadonnées. La première représente la hiérarchie de types du programme et l'autre associe à chaque objet monitoré un type. Par défaut, ces métadonnées sont déjà produites par les compilateurs. Les RunTime Type Information (RTTI) encodent la hiérarchie de types et les *vtables* (*i.e.* des tableaux de pointeurs vers les méthodes des objets) permettent d'identifier de manière unique le type d'un objet lors de l'exécution. Actuellement, l'implémentation de ces métadonnées est spécifique à chaque compilateur. Celles-ci sont de qualités inégales et c'est pour cela que les RTTI souffrent d'une mauvaise réputation. De plus, le standard C++ limite l'utilisation de ces métadonnées aux seuls types polymorphes. Aussi, les outils CaVeR et TypeSan ont fait le choix de les remplacer par les leurs.

Au final, cette approche n'est en rien nouvelle car c'est exactement le comportement par défaut du compilateur lorsqu'il rencontre un `dynamic_cast`. La valeur ajoutée des *sanitizers* est pourtant multiple. Tout d'abord, l'ajout de la vérification s'applique à toutes conversions potentiellement dangereuses de manière systématique. De plus, il n'est pas nécessaire de modifier le code source initial, ce qui facilite l'adoption de ce genre d'outil.

La principale amélioration qu'apporte les *sanitizers* réside dans leur capacité à détecter un plus grand panel de confusions. Ubsan, le premier outil à être apparu, réalise un *simple* remplacement du `static_cast` par sa propre implémentation optimisée du `dynamic_cast`. Ceci permet de détecter uniquement les confusions entre types polymorphes, c'est-à-dire des types possédant une *vtable*. CaVeR étend la classe de bug détectée aux types non polymorphes grâce à de nouvelles métadonnées. Par la suite, Haller *et al.* [6] mettent en évidence de potentiels faux négatifs avec CaVeR. En effet, celui-ci ne se préoccupe que des objets alloués sur le tas à l'aide de l'opérateur *new*. Or une confusion de type peut aussi s'appliquer à un objet sur la pile d'appel ou une variable globale. TypeSan étend alors les capacités du *sanitizer* afin de considérer l'ensemble des variables.

Chaque outil a su proposer des améliorations pour réduire le coût de la vérification dynamique. En effet, l'instrumentation du code impose un ralentissement de l'exécution. Ce coût est encore trop élevé pour que les *sanitizers* puissent être considérés comme une véritable contre-mesure à utiliser en production. Toute amélioration des performances serait une contribution intéressante à la fois pour les *sanitizers* et pour les compilateurs eux-mêmes qui pourraient les intégrer dans leur implémentation des `dynamic_cast`.

Les *sanitizers* sont souvent comparés à d'autres contre-mesures cherchant à protéger l'intégrité des *vtables* ou à assurer l'intégrité du flot de contrôle d'un programme ([6], [5]). En effet, la corruption d'un pointeur de *vtable* qui mène au détournement du flot d'exécution lors d'un appel à une méthode virtuelle est un événement qui peut être détecté par les trois vérifications précédémment énoncées.

Ubsan, CaVeR et TypeSan ne cherchent qu'à identifier les confusions de type entre des classes C++. Cependant, les confusions de type entre types primitifs ou des structures C existent et peuvent être tout aussi exploitables [7].

## III. Analyse de binaires

Malgré leur efficacité les *sanitizers*, leur utilisation s'avère limité par la recompilation qu'implique ces outils. Dans la majorité des cas lors d'un audit de sécurité, le code source, les dépendances et l'environnement de développement ne sont pas fournis ou seulement partiellement, rendant la compilation du logiciel impossible. Actuellement, il n'existe pas l'équivalent des *sanitizers* visant à détecter les confusions de type à partir du binaire. Les travaux qui s'en rapprochent le plus s'intéressent à l'intégrité du flot de contrôle des méthodes virtuelles en C++, appelée aussi *dynamic dispatch*. Ce mécanisme est la cible privilégiée des attaquants car il peut être détourné facilement à l'aide d'une corruption mémoire, modifiant ainsi le flot de contrôle à leur avantage.

Dewey et Giffin ont proposé une méthode pour trouver cette conséquence particulière d'une confusion de type au sein d'un binaire [8]. Elle repose sur la reconnaissance des appels aux méthodes virtuelles et sur la reconstruction des classes (leur structure interne et leurs méthodes virtuelles). L'analyse de détection proposée cherche à calculer pour chaque appel à une méthode virtuelle l'ensemble des classes d'objets qui peuvent réaliser cet appel. De plus, une autre analyse vérifie que toutes les classes respectent certaines propriétés en termes de structure interne et de nombres de méthodes virtuelles.

Ces travaux souffrent d'une limitation majeure : leur analyse est statique et interprocédurale. Or, le calcul de l'ensemble des classes possibles pour un pointeur statique est un problème NP-complet comme l'a démontré Pande [2]. Ceci les oblige à travailler sur des versions *simplifiées* de programmes vulnérables. Ainsi, ils limitent la taille du graphe d'appel en rapprochant le point d'entrée du programme de la fonction vulnérable et suppriment également les régions de code non liées au bug. C'est une forme de *slicing* manuel, permettant de restreindre l'analyse à une sous-portion vulnérable du binaire.

L'outil vfGuard [9] propose une autre solution. Il assure le contrôle du flot d'intégrité des méthodes virtuelles d'un binaire au cours de son exécution. Sa détection englobe à la fois les corruptions inhérentes au programme dues à une confusion de type, ainsi que celles dues à une corruption mémoire. L'approche de vfGuard combine une analyse statique, moins poussée que [8] à des vérifications dynamiques des appels de méthodes virtuelles. Il s'agit de récupérer l'ensemble des pointeurs de fonction des *vtables* et l'ensemble des appels aux méthodes virtuelles. Ensuite, à chaque appel est associé un ensemble de cibles (des pointeurs de fonction) représentant le flot de contrôle *normal* du programme.

Le standard C++ assure que pour un appel de méthode virtuelle, l'ensemble des cibles possibles est un sous-ensemble de l'union des *vtables* de toutes les classes d'objets réalisant cet appel. Par conséquent, l'ensemble de toutes les méthodes virtuelles toutes classes confondues est une approximation sûre mais peu précise. Pour affiner cet ensemble pour chaque appel, vfGuard utilise des heuristiques, comme le respect d'une même convention d'appel par exemple.

Ces deux travaux peuvent seulement détecter un effet de bord de la confusion de type, le détournement de flot de contrôle à travers la corruption d'un pointeur vers une *vtable*. Cependant, ils n'aident en rien dans les autres situations menant à des fuites d'information ou de la corruption mémoire, ni à la détection de l'origine du bug, ce qui est un des points forts des *sanitizers*.

Deux problèmes majeurs ressortent de l'état de l'art : la décompilation de binaires compilés depuis le langage C++ et les performances de l'instrumentation de binaire.

Le cœur des travaux présentés consiste à proposer des analyses permettant de retrouver certains artéfacts C++. La décompilation du langage C++ est un problème ouvert en soi et il existe une littérature abondante à ce sujet ([10], [11], [12], [13]) ainsi que des outils comme le décompilateur d'IDA Pro HexRays [14] ou le décompilateur snowman [15].

Le second problème est lié aux performances de l'instrumentation. Par exemple, les auteurs de vfGuard [9] ont envisagé des solutions fondées sur la réécriture de code binaire afin de limiter l'ajout de code aux seuls endroits nécessaires. Ces solutions consistant à modifier le binaire directement se sont révélées trop immatures pour traiter des binaires gros et complexes. Nous avons aussi été confrontés à cette difficulté lors de nos tests sur Chrome.

## IV. Preuve de concept

Le but de nos travaux est de montrer qu'il est possible de mettre en place la méthode des *sanitizers*, en particulier celle de Ubsan, pour la détection de confusion de type dans le cadre d'une analyse de code binaire. Pour se faire, nous avons décidé de réalise une preuve de concept sur un exemple réel, la CVE-2013-0912 au sein de Chrome sur la plateforme Windows.

Dans un premier temps, nous mettons de côté le problème de décompilation et de la reconstruction de la hiérarchie de type. Nous supposons donc qu'un effort de rétro-ingénierie a été produit avant notre analyse pour fournir les types de haut niveau. Dans notre cas, nous utilisons les symboles de debug complets pour obtenir ces informations. A ce stade, nous possédons la même connaissance du système de type qu'une analyse se basant sur le code source. Cependant, il nous manque encore une information cruciale: la situation des opérations de conversion au sein du code binaire. Ces opérations disparaissent totalement lors de la compilation, il nous faut alors les reconstruire. Pour cela, nous utilisons les capacités de propagation de types du décompilateur HexRays, pour inférer les conversions entre types de haut niveau. Grâce à notre connaissance de la hiérarchie de type, il nous est possible de distinguer les conversions sûres de celles potentiellement dangereuses. A la fin de cette étape, nous obtenons une liste d'instructions à surveiller lors de l'exécution du programme et le type attendu à l'issu de chaque conversion. Ceci clos la phase d'analyse statique.

Pour la seconde phase de l'analyse, notre vérification dynamique doit être capable de retrouver le type réel du pointeur d'objet qui est converti. Malheureusement, nous ne pouvons pas utiliser les RTTI ou un autre type de métadonnée stockant de l'information sur le type dynamique des objets, comme le

ferait un *sanitizer* ayant instrumenté le code source. En effet, Chrome n'est pas compilé avec les RTTI par défaut et c'est le cas de nombreux autres projets. Cependant, notre analyse comme celle de Ubsan se limite aux types polymorphes. Il est donc possible d'utiliser la *vtable* de chaque type polymorphe pour l'identifier de manière unique Notre vérification dynamique commence donc par récupérer l'adresse de la *vtable* stockée au sein même de l'objet. Cette adresse est stockée toujours au même endroit. Puis, nous utilisons les symboles de debug pour retrouver le type de la classe correspondante. Le type dynamique est ensuite comparé avec le type attendu statiquement en respectant la hiérarchie de type.

Cette vérification doit être appliquée à l'ensemble de la liste d'instructions produite durant la première phase statique. Il existe plusieurs solutions aujourd'hui pour instrumenter un binaire: les API de debug, la Static Binary Instrumentation (SBI), la Dynamic Binary Instrumentation (DBI), l'émulation ou la virtualisation. Notre choix a été guidé par: (1) la nature du binaire, Chrome s'exécute en espace utilisateur, il n'est pas *malicieux* (absence de techniques d'anti-debug) mais sa taille est importante et (2) les caractéristiques de la vérification dynamique. Celle-ci est ciblée, car nous nous intéressons qu'à certaines instructions dont l'adresse est connue avant l'exécution du programme. De plus elle est aussi ponctuelle, car la connaissance du contexte d'exécution à un instant donné est suffisante. Le premier critère élimine d'emblée la solution d'un émulateur ou d'un hyperviseur modifié car leur intérêt principal est de pouvoir instrumenter du code noyau. Théoriquement, la SBI remplie parfaitement notre second critère car elle réduit au minimum le nombre d'instructions rajoutées au binaire et ne dépend pas d'interruptions systèmes pour interrompre l'exécution du programme en cours, ce qui est optimal en ce qui concerne les performances. Nous avons essayé d'utiliser l'outil Dyninst [16] pour réaliser cette tâche mais celui-ci n'est pas suffisamment robuste actuellement pour traiter notre cible. Faute d'autres alternatives disponibles, nous avons donc considéré les deux autres solutions.

La DBI est une méthode populaire de nos jours pour l'analyse de binaire, ses performances sont bien meilleures que celles d'un debugger lorsqu'il est nécessaire d'instrumenter un binaire pas à pas. En effet, son mécanisme de traduction du code binaire à la volée permet d'éviter l'utilisation d'interruption à chaque instruction ce qui est le mode de fonctionnement classique d'un debugger. Cependant, cette méthode ne semble pas adaptée à notre problème. Pour fonctionner la DBI doit avoir une vue intégrale des instructions exécutées pour pouvoir insérer le code de l'instrumentation. Cela induit un ralentissement sur l'ensemble des instructions même celles où aucune vérification n'est nécessaire. Or pour un binaire de plusieurs milliards d'instructions cela est rédhibitoire. Nos tests confirment cette intuition, l'impact de la DBI que ce soit avec Pin [17] ou DynamoRIO [18] est clairement perceptible de par la lenteur de l'affichage des fenêtres et du traitement des entrées utilisateurs.

Finalement, nous nous sommes tournés vers la méthode la plus ancienne les debuggers. Les *breakpoints* qu'ils proposent permettent de stopper l'exécution du programme uniquement aux seules instructions désirées, ce qui correspond exactement à l'instrumentation souhaitée pour la vérification dynamique. Seul la question des performances des *breakpoints* peut être problématique dans notre cas. Néanmoins, le nombre de breakpoints nécessaires par rapport au nombre total d'instructions au sein du binaire est de 0.5%. Cela signifie que la majorité du temps, le programme s'exécute de manière native. Nous avons donc automatisé WinDBG pour positionner automatiquement des *breakpoints* sur l'ensemble des instructions potentiellement dangereuses. A chaque arrêt la vérification sur le type dynamique est effectué, si celle-ci est incorrecte, le processus reste suspendu. Cette solution simple a suffit pour obtenir le résultat escompté avec des performances acceptables, c'est-à-dire en restant dans l'ordre de grandeur d'une exécution sans instrumentation (du niveau de la minute).

## V. Conclusion

Le problème de la détection de la confusion de type met en valeur l'écart entre analyse de code source et de binaire. De par sa nature, ce problème nécessite la connaissance de la hiérarchie de classes et la localisation des conversions de types, deux propriétés propres au code source et difficiles à reconstruire depuis le binaire. Cet écart est aussi visible dans les pistes d'améliorations; l'analyse de code binaire lutte encore pour obtenir les informations nécessaires pour des analyses plus poussées, tandis que l'analyse de code source vise à rendre toujours plus performante la vérification dynamique.

Selon nous, l'analyse de code binaire pour la détection de confusion de type devrait tendre vers l'approche des *sanitizers*. Nous avons vu qu'une approche purement statique était vouée à l'échec. De plus, les avancées algorithmiques proposées par TypeSan sont orthogonales aux problèmes rencontrés par l'instrumentation de binaire. Par conséquent, les prochains *sanitizers* de binaires auront tout intérêt à les intégrer dans leur vérification dynamique.

Il n'est cependant pas réaliste de vouloir atteindre le niveau actuel de détection de TypeSan. Une première étape serait de viser les capacités d'Ubsan, c'est-à-dire détecter les confusions entre types polymorphes avec l'aide des RTTI.

Nos travaux actuels tentent d'appliquer l'approche des *sanitizers* à des binaires. Ils reposent sur la propagation de types du décompilateur HexRays d'IDA Pro pour reconstruire une partie des conversions disparues et identifier celles dangereuses. La vérification dynamique est assurée par le debugger WinDBG, cette solution s'est avéré bien adaptée à nos besoins et suffisamment robuste pour surveiller l'exécution du navigateur Chrome.

Nos premiers tests utilisent les symboles de debug pour pallier le manque d'information sur les types. Une piste d'amélioration serait de supprimer cette contrainte en remplaçant les symboles par des heuristiques de reconstruction de types C++ de l'état de l'art, par exemple: Snowman [15].

## REFERENCES


[1] M. Labs, 2013, https://labs.mwrinfosecurity.com/blog/mwr-labs-pwn2own-2013-write-up-kernel-exploit/.

[2] H. D. Pande and B. G. Ryder, "Static type determination for C++," in *C++ Conference*. USENIX Association, 1994, pp. 85–98.

[3] B. Stroustrup and H. Sutter, "C++ core guidelines," https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#exception-22.

[4] "UndefinedBehaviorSanitizer (UBSan) - The Chromium Projects." [Online]. Available: http://www.chromium.org/developers/testing/undefinedbehaviorsanitizer

[5] B. Lee, C. Song, T. Kim, and W. Lee, "Type casting verification: Stopping an emerging attack vector," in *24th USENIX Security Symposium*. USENIX Association, 2015, pp. 81–96.

[6] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe, "TypeSan: Practical Type Confusion Detection," in *ACM SIGSAC'16*. ACM Press, 2016, pp. 517–528.

[7] F. Saudel, "Confusion de type en c++ : la performance au détriment de la type safety," in *SSTIC*, 2016.

[8] D. Dewey and J. T. Giffin, "Static detection of C++ vtable escape vulnerabilities in binary code." in *NDSS*. The Internet Society, 2012.

[9] A. Prakash, X. Hu, and H. Yin, "vfguard: Strict protection for virtual function calls in cots c++ binaries," in *NDSS'15*. The Internet Society, 2015.

[10] K. Yoo and R. Barua, "Recovery of Object Oriented Features from C++ Binaries," in *APSEC'14*, vol. 1. IEEE, 2014, pp. 231–238.

[11] V. Srinivasan and T. W. Reps, "Recovery of class hierarchies and composition relationships from machine code," in *CC, ETAPS'14*, ser. Lecture Notes in Computer Science, A. Cohen, Ed., vol. 8409. Springer, 2014, pp. 61–84.

[12] W. Jin, C. Cohen, J. Gennari, C. Hines, S. Chaki, A. Gurfinkel, J. Havrilla, and P. Narasimhan, "Recovering C++ objects from binaries using inter-procedural data-flow analysis," in *ACM SIGPLAN'14*. ACM, 2014, pp. 1:1–1:11.

[13] J. Caballero and Z. Lin, "Type inference on executables," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 65:1–65:35, 2016.

[14] "IDA PRO decompiler's Hexrays," https://www.hex-rays.com/products/decompiler/index.shtml.

[15] Y. Derevenets, "Snowman decompiler," https://derevenets.com/.

[16] L. C. Harris and B. P. Miller, "Practical analysis of stripped binary code," *SIGARCH'05*, vol. 33, no. 5, pp. 63–68, 2005.

[17] C. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM SIGPLAN'05*, 2005, pp. 190–200.

[18] D. Bruening, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004.